Calhoun
Institutional Archive of the Naval Postgraduate School

Calhoun: The NPS Institutional Archive

DSpace Repository

Theses and Dissertations 1. Thesis and Dissertation Collection, all items

2011-03

Using statechart assertion for the formal validation and verification of a real-time software system a case study

Beylin, Konstantin (Chris).

Monterey, California. Naval Postgraduate School

http://hdl.handle.net/10945/5818

This publication is a work of the U.S. Government as defined in Title 17, United States Code, Section 101. Copyright protection is not available for this work in the United States.

Downloaded from NPS Archive: Calhoun

DUDLEY KNOX LIBRARY

http://www.nps.edu/library

Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

USING STATECHART ASSERTION FOR THE FORMAL VALIDATION AND VERIFICATION OF A REAL-TIME SOFTWARE SYSTEM: A CASE STUDY

by

Konstantin (Chris) Beylin

March 2011

Thesis Co-Advisors: Doron Drusinsky
Man-Tak Shing

Approved for public released; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** March 2011	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE Using Statechart Assertion for the Formal Validation and Verification of a Real-Time Software System: A Case Study			**5. FUNDING NUMBERS**
6. AUTHOR(S) Konstantin (Chris) Beylin			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number _______NA_______.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public distribution; distribution is unlimited			**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

Verification and validation (V&V) is one of the software engineering disciplines that helps build quality into software. V&V comprehensively analyzes and tests software to determine that it performs its intended functions correctly, and ensures that it does not perform unintended functions. However, V&V traditionally relies on manual examination of software requirements, design artifacts and the systematic or random testing of target code. As software-intensive systems become increasingly complex, traditional V&V techniques are inadequate for locating subtle errors in the software. It is even more challenging to test embedded real-time systems characterized by temporal behavior. For the past several decades, academia has actively researched the use of formal methods that help improve the quality of the software. Nonetheless, the techniques developed using formal methods still are not widely accepted in industry and in government.

Professor Doron Drusinsky from Naval Postgraduate School (NPS) has developed a novel lightweight formal specification, validation and verification technique. The technique is focused on modeling reactive real-time systems with UML-based formal specifications and log file based Runtime Verification (RV).

This thesis presents a case study as a proof of concept in support of this V&V technique, applied on a complex, already developed and fielded mission-critical system. It has successfully demonstrated a pragmatic approach in achieving a high quality V&V testing.

14. SUBJECT TERMS Formal specification, validation, verification, statechart assertion, real-time, reactive, testing, UML, runtime verification			**15. NUMBER OF PAGES** 91
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

USING STATECHART ASSERTION FOR THE FORMAL VALIDATION AND VERIFICATION OF A REAL-TIME SOFTWARE SYSTEM: A CASE STUDY

Konstantin (Chris) Beylin
Civilian, United States Navy
BS., California State University, Northridge, 1990

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN SOFTWARE ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
March 2011

Author: Konstantin (Chris) Beylin

Approved by: Doron Drusinsky
Thesis Co-Advisor

Man-Tak Shing
Thesis Co-Advisor

Peter J. Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Verification and validation (V&V) is one of the software engineering disciplines that helps build quality into software. V&V comprehensively analyzes and tests software to determine that it performs its intended functions correctly, and ensures that it does not perform unintended functions. However, V&V traditionally relies on manual examination of software requirements, design artifacts and the systematic or random testing of target code. As software-intensive systems become increasingly complex, traditional V&V techniques are inadequate for locating subtle errors in the software. It is even more challenging to test embedded real-time systems characterized by temporal behavior. For the past several decades, academia has actively researched the use of formal methods that help improve the quality of the software. Nonetheless, the techniques developed using formal methods still are not widely accepted in industry and in government.

Professor Doron Drusinsky from Naval Postgraduate School (NPS) has developed a novel lightweight formal specification, validation and verification technique. The technique is focused on modeling reactive real-time systems with UML-based formal specifications and log file based Runtime Verification (RV).

This thesis presents a case study as a proof of concept in support of this V&V technique, applied on a complex, already developed and fielded mission-critical system. It has successfully demonstrated a pragmatic approach in achieving a high quality V&V testing.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I. INTRODUCTION....................1
A. MOTIVATION....................1
B. PURPOSE....................3

II. BACKGROUND....................5

III. OVERVIEW OF MIDS CONTROLLER SUT....................17
A. SYSTEM ARCHITECTURE....................19
1. Subsystems Brief Description....................19
B. LINK-16 MESSAGES OVERVIEW....................20
C. NL REQUIREMENTS....................21
1. MC Power Up Initialization Sequence....................21
2. HHO Requirements....................22

IV. SPECIFICATION AND VALIDATION PHASE....................25
A. STATECHART ASSERTIONS....................25
B. VALIDATING MC POWER UP INITIALIZATION SEQUENCE....................32
C. VALIDATING HHO (HARM HAND OFF)....................43

V. VERIFICATION PHASE....................51

VI. CONCLUSION....................69

LIST OF REFERENCES....................71

INITIAL DISTRIBUTION LIST....................73

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

Figure 1: Coverage space. From [3]. 3
Figure 2: Cost space. From [3]. 3
Figure 3: Software development activities and testing levels—the "V Model." From [6]. 5
Figure 4: Relative cost versus development phase. From [7]. 7
Figure 5: Scenario for ambiguous requirement. From [9]. 12
Figure 6: Program modeled as graph. 12
Figure 7: Link-16 participants' implementation. From [14]. 18
Figure 8: System architecture 19
Figure 9: Finite state machine diagram. From [11]. 26
Figure 10: Statechart superstate. From [11]. 26
Figure 11. Orthogonality example. From [11]. 27
Figure 12: State diagram without statechart orthogonality. From [11]. 27
Figure 13: General syntax of a statechart transition. From [9] 29
Figure 14: Statechart assertion diagram for R1 requirement 29
Figure 15: Statechart assertion diagram for R2 requirement 30
Figure 16: Statechart assertion diagram for R3 requirement. From [9]. 31
Figure 17: MC Validation and Verification with assertion repository 32
Figure 18: Statechart assertion diagram for MC Power Up Initialization Sequence 35
Figure 19: Validation test results (all test cases passing) 38
Figure 20: Validation test results (test cases 4 failed) 39
Figure 21: Timeline diagram rendering of a validation test expecting a success. From [16]. 39
Figure 22: Timeline diagram rendering of a validation test expecting a failure. From [16]. 40
Figure 23: Animation for test case 1 41
Figure 24: Animation for the test case 2 42
Figure 25: Statechart assertion diagram for HHO 45
Figure 26: Timeline diagram for test case 1 47
Figure 27: Invalid statechart assertion for R5 requirement. From [15] 48
Figure 28: Valid statechart assertion for R5 requirement. From [15]. 49
Figure 29: Lab setup for SUT verification 52
Figure 30: Instrumented MC code 54
Figure 31: Namespace mapping 61
Figure 32: Verification test results for MC Power Up Initialization Sequence 62
Figure 33: MC Power Up Initialization Sequence with failed assertion and unmatched events 63
Figure 34: Coverage animation from verification test for failed MC Power Up Initialization Sequence 64
Figure 35: Timeline diagram for verification test (Power Up Initialization Sequence) 65
Figure 36: Verification test results for HHO 66
Figure 37: Assertion repository viewer for HHO 66
Figure 38: Animation coverage from verification test for HHO 67

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF LISTINGS

Listing 1: C++ function calculates number of zeros from input parameter. After [6]. 8
Listing 2: Validation test 36
Listing 3: Test case for time constraint violations 37
Listing 4: Test case for out of order sequencing violation 38
Listing 5: Validation test case 1 46
Listing 6: Validation test case 2 46
Listing 7: Validation test case 3 47
Listing 8: Validation test case with the same result. From [15]. 49
Listing 9: Validation test case with different results. From [15]. 50
Listing 10: Code snippet for instrumented PeriodicTask function 55
Listing 11: Instrumentation is placed prior to SemTake system call (incorrect) 56
Listing 12: Instrumentation is placed after SemTake system call (correct) 57
Listing 13: Log file snippet for MC Power Up Initialization Sequence 58
Listing 14: Log file snippet for HHO 59
Listing 15: Verification JUnit test for MC Power Up Initialization Sequence 60
Listing 16: Verification JUnit test for HHO 60

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

CMC	Central Mission Computer
DSMU	Data Storage Memory Unit
GUI	Graphical User Interface
HARM	High Speed Anti-Radiation Missile
HHO	HARM Hand Off
IM	Information Manager
MC	MIDS Controller
MIDS	Multifunctional Information Distribution System
MM	Mission Manager
SUT	System Under Test
TDS	Tactical Display Computer
TJHS	TADL J Host Simulator
TADL	Tactical Digital Information Link
V&V	Verification and Validation
VME	VERSAmodule Eurocard
XML	Extensible Markup Language

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank my beautiful wife, Irina, and my children, Nicholas and Natalie, for their patience, sacrifice and unconditional love. Your support has kept me motivated and focused throughout this process.

This thesis would not have been possible without the guidance and the help of Professor Doron Drusinsky, who offered invaluable assistance and allowed me to use his software application.

Special thanks to my management, Balaji Iyer and Thaddeus Falarski, at Naval Air Warfare Center Weapons Division, for their active support by providing financial means and their commitment to my continuous education.

Finally, I would like to extend my gratitude to all those who contributed to editing, proofreading and processing this thesis; especially Professor Man-Tak Shing and Steve Fisher.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. MOTIVATION

Over the past several decades, software has become an intrinsic part of our industry, as well as of our daily life. Although industry spends billions of dollars each year developing software, many software systems fail to satisfy their users. There are many examples of the impact of software failures on our daily lives: an Air Traffic Control system failure in the main Los Angeles airport, disrupting over 800 flights across the country; the crash of the $125 million NASA Mars Orbiter; the loss of the European Ariane 5 rocket and its payload of satellites; the loss of life in the Middle East due to numerical error by the Patriot missile system software. A study conducted in 2002 and sponsored by the National Institute of Standards and Technology (NIST) found that the annual cost of software errors to the U.S. economy is approximately $59.5 billion, which in 2002 was about 0.6 of the Gross Domestic Product [1]. This study also pointed out that significant improvement in software testing infrastructure could drastically reduce annual cost of software errors, to approximately $22.2 billion per year, a reduction of more than half the cost. Without any doubt, software verification and validation are instrumental activities on any project, with the primary goal of reducing software errors and maximizing software quality. Unfortunately, due to the inherent complexity of software, an exhaustive testing of all execution paths is practically infeasible; that is, testing can never be complete. In general, testing can show only the presence of errors, not their absence. In a typical software development project, the cost of providing assurance that the software will perform satisfactorily in terms of its functional and nonfunctional specifications within the expected deployment environments via appropriate debugging, testing, and verification activities can easily range from 30 to 50 percent of the total development cost. Then, in spite of these costly efforts, some software errors will remain in the final product to be discovered by its users.

According to IEEE Standard Software V&V [2], *validation* is defined as a process of evaluating a system or components during or at the end of the development process to determine whether a system or components satisfied a specified requirement, and

verification is defined as the process of evaluating a system or components to determine whether a system of a given development phase satisfies the condition imposed at the start of that phase. In short, traditional software engineering distinguishes between validation, a process for getting the right product, and verification, a process for getting the product right. However, V&V is typically conducted as a single phase of a development process right after the product has been developed and has undergone unit, integration and system testing. The danger in this approach is the possibility of discovering that the product does not fulfill its specific intended purpose, at very late stages in the development process. Also, validating the temporal behavior of a set of requirements that contains timing and time-series constraints could be a very challenging task. In particular, it is extremely difficult to validate temporal requirements of embedded systems that do not have any external display capabilities.

Formal Validation & Verification (FV&V) of reactive systems has been actively researched for the last three decades. Nevertheless, FV&V techniques have not been widely adopted by industry or government even for use in safety-critical commercial and defense applications [3]. In [3], the authors describe a three-dimensional tradeoff space that qualitatively compares model checking, theorem proving and Runtime Verification (RV), which is also known as Execution-Based Model Checking (EMC) in three aspects: specification, association with the underlying SUT, and verification. The coverage cube, shown in Figure 1, represents the coverage-space tradeoff between three FV&V techniques. The cost cube, shown in Figure 2, represents the cost in each dimension induced by a given FV&V technique. From Figure 1 and Figure 2, it is evident that RV presents a viable solution. The FV&V paper [3] advocates the use of RV due to its being both cost-effective and coverage-effective in the specification/validation dimension.

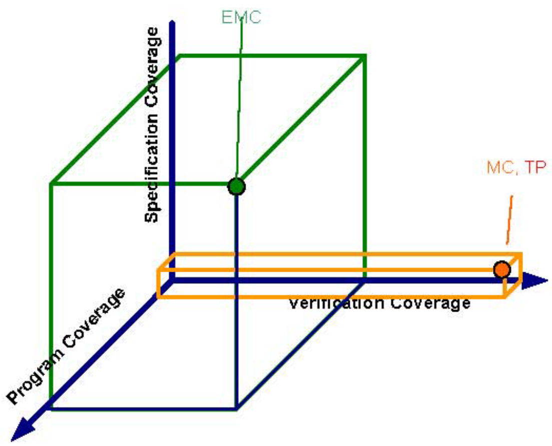

Figure 1: Coverage space. From [3].

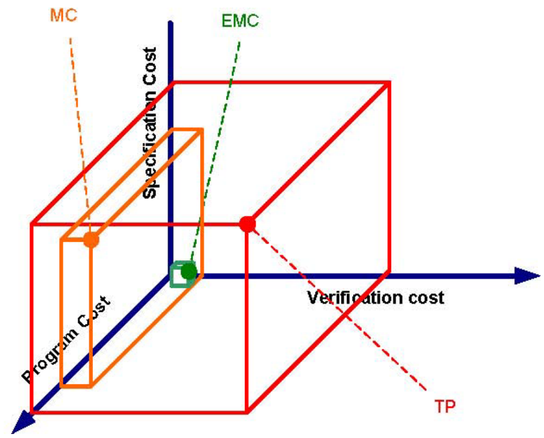

Figure 2: Cost space. From [3].

This thesis will research alternative validation and verification techniques. The research will be based on the paper published by Professor Drusinsky [4]. The paper proposes a more coherent approach to V&V by using statechart assertions. In fact, the above IEEE validation definition is really a union of the proposed verification and validation. Statechart assertions are used to assist in firming up the Natural Language (NL) requirements and expressing them in formal specification. Subsequently, validation testing assures that each formal specification conforms to the software requirements, and verification testing assures that the System Under Test (SUT) conforms to the formal specification assertions [4].

B. PURPOSE

This thesis attempts to answer the following questions:

1. Is it feasible to capture Natural Language (NL) requirements into formalized specification and subsequently validate these requirements?

2. Is it possible for the V&V team to really have an independent, fresh view of the system that matches the developers' cognitive intent of the requirement?

3. Is it possible to verify the system's sequential, temporal, and propositional behavior while running on an actual target system?

An additional purpose is to develop a proof of concept demonstration for the V&V process conducted on the real operational software system using the statechart assertion formalism. The StateRover (developed by Professor Drusinsky) is based on the Eclipse development framework, and is a commercially available tool that fully supports statechart assertion validation and verification. The real-world application that will be used in this thesis is the Multifunctional Information Distribution System (MIDS) Controller (MC).

II. BACKGROUND

The concept of V&V emerged in the late 1960s and 1970s as the use of software in military and nuclear power systems increased. Verification involves evaluating software during each life-cycle phase to ensure that it meets the requirements set forth in the previous phases. Validation involves testing software or its specification at the end of the development effort to ensure that it meets its requirements [5]. A different level of testing accompanies each distinct software development activity. Figure 3 illustrates the "V Model" of a typical scenario for testing levels and how the relate to software development activities by isolating each step.

This thesis focuses on verification and validation testing, which can be applied to any level of testing. But typically, verification and validation is closely tied to an acceptance test. An acceptance test is designed to determine whether the completed software in fact meets its requirement specification, which involves both validation and verification. In other words, acceptance testing ensures that the software does what the user specifies. As a result, acceptance testing must involve individuals who have strong system domain knowledge.

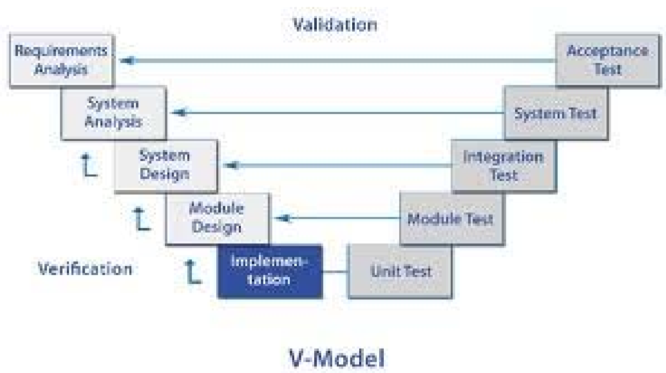

Figure 3: Software development activities and testing levels—the "V Model." From [6].

Amman and Offutt [6] point out a different level of skill set necessary for validation and verification. They indicate that validation usually depends on domain knowledge, that is, knowledge of the specification for which the software is

developed [6]. Verification is usually a more technical activity that uses knowledge about the individual software artifacts, requirements, and specification [6].

As mentioned in the Chapter I, one of the important limitations of software testing is that it can show only the presence of failures, not their absence. This is indeed a fundamental, theoretical limitation; generally speaking, the problem of finding all failures in a software program is undecidable. Given the fact that, realistically, V&V activity is bound to a certain amount of scheduled time and resources, and also that it will not ever guarantee error-free software with 100% certainty, there is emerging demand for high quality V&V testing.

Previously-conducted studies indicate that the highest numbers of software errors are due to failures in software requirements specification. The 1992 study of Voyager/Galileo software revealed that out of 197 significant or catastrophic errors, 194 errors were contributed to by specification errors of functions and interfaces, and only 3 were caused by coding errors [7]. Furthermore, fixing errors during specification phase is significantly less costly than in subsequent phases [7]. Figure 4 depicts relative cost associated with fixing software errors in various development phases. Without a doubt, finding software requirement specification errors early in the development cycle is the key to higher quality software. Another important factor besides finding software specification errors is V&V independence. It is important to have an independent validation and verification team because it can formulate its own understanding of the problem. This is because technical independence has a greater chance of detecting subtle errors, which were overlooked by developers, who are too close to the actual implementation [8].

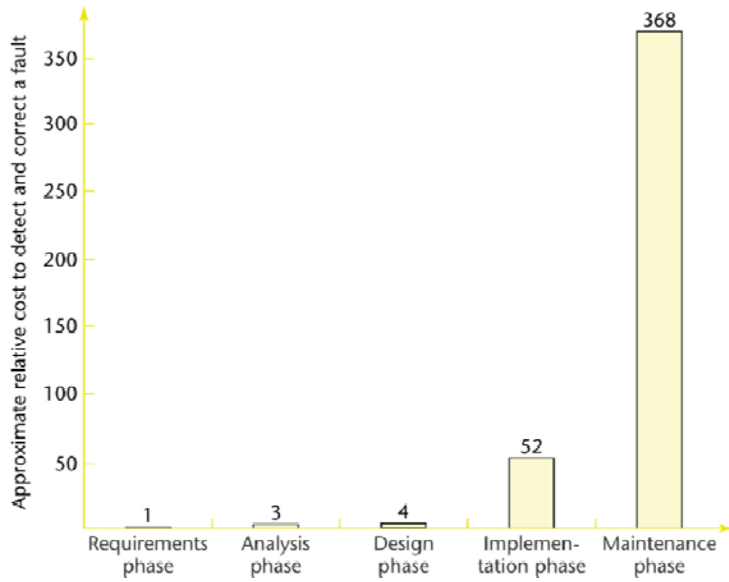

Figure 4: Relative cost versus development phase. From [7].

It is important to identify and point out a clear distinction in testing terminology.

- Definition: *Software Fault*: A static defect in the software. This usually refers to any mistake made by humans. It could be related to design, implementation (coding), wrong interpretation of specification, or specification omission.
- Definition: *Software Error*: An incorrect internal state that is the manifestation of some fault. This refers to an incorrect software state due to an inherent software fault.
- Definition: *Software Failure*: External, incorrect behavior with respect to the requirements or other description of the expected behavior. This refers to the actual software failure exposed during runtime [6].

It is important to understand the distinction between software error and software failure. For example, a software error could occur but might not manifest or propagate to become a software failure. Listing 1 illustrates example of software error and failure.

```cpp
int numZero(int* x, int length)
{
  int count = 0;
  for (int i=1; i< length; i++)
  {
    if (x[i] == 0)
    {
      count++;
    }
  }
  return count;
}
```

Listing 1: C++ function calculates number of zeros from input parameter. After [6].

Obviously, the fault in this function is that is starts looking for zeros at index 1, instead of at index 0, as is necessary for arrays in C++. As a result, input to numZero with an array x={3,5,0} evaluates to 1, while another input with an array x={0,8,5} evaluates to 0. In both of these cases the fault is executed, only the second case results in error, which may lead to failure. The point is that a software error could be encountered without causing the program results to be incorrect [6]. The distinction lies in error propagation or manifestation to cause incorrect behavior as oppose to be still incorrect but not externally visible.

Analyzing the clear distinction between fault, error and failure leads to a test model, which states that three conditions must be present for a software failure to be observed:

1. The locations in the program that contains the fault must be reached (*Reachability*).

2. After executing the location, the state of the program must be incorrect (*Infection*).

3. The infected state must propagate to cause some output of the program to be incorrect (*Propagation*) [6].

This “RIP” model is indeed very important for any test coverage criterion. By definition, a coverage criterion is a collection of rules that impose test requirements on test cases.

Given the fact that software testing can never be complete, and it is not possible to test every permutation of every execution path, coverage criterion provides practical rules of when to stop testing. Coverage criterion is simply a recipe for generating test requirements in a systematic way. In this case, test requirements could be any software artifact that test case must satisfy or cover. For example, test requirements could be source code, design components, specification modeling elements, or an FSM (Fine State Machine) diagram.

Real-time software system must respond to externally generated input stimuli within a finite and specified period. Real-time systems usually have explicit time constraints that specify the response time and temporal behavior. Real-time systems are typically constrained by hard time requirements and sequencing logic. A time constraint on the response time of a request is called a deadline. For example, in a nuclear power plant controller, a requirement could be: Activate shut down sequence within 500 microseconds, upon detection of radiation meltdown. Timeliness refers to the ability of software to meet time constraints.

Real-time systems are often characterized as reactive, with this reactive system event-driven, that is, continuously having to interact with external and internal stimuli. Examples include avionics systems, car engine and brakes, industrial control systems, a satellite flight manager, air traffic management, telecom switches. Reactive systems perform an ongoing and often never-ending computation, in which each calculation uses information generated by previous invocations [9]. Thus all reactive systems have a notion of a state and a state transition. In contrast, transformational components are stateless; that is, they have no memory that persists between successive invocations of the component. For example, converting temperature from Fahrenheit to Celsius is a transformational component. Transformational systems perform a fresh computation every time they are invoked [9]. Reactive systems are typically defined by sequencing behavior that consists of sequencing events, conditions, constraints on data values, and timing. Sequencing logic in general specifies the order and sequencing of particular events: Sequencing behavior has two types of possible constraints.

Timing constraints describe the timely start and termination of successful computations at a specific point of time, such as the deadline of a periodic computation or the maximum response time to handle a certain event [8].

Time-series constraints describe the timely execution of a sequence of data values within a specific duration of time [8]. For example, specification with time series: No more than five sampling events are allowed within a one-second time interval following receipt of an RF signal.

Software fault, error and failure with respect to real-time systems are defined with similar resemblance to previous definitions, but in addition introducing time and sequence elements as follows:

Timeliness fault denotes a mistake in the implementation or specification of a real-time system that may result in incorrect sequencing behavior. For example, a misunderstood specification could result in an out of order sequence of events implementation dealing with a landing space shuttle.

Timeliness error occurs when the system internally deviates from assumptions about it temporal behavior. Timeliness errors are extremely difficult to detect without extensive logging and precise knowledge about the internal behavior of the system.

Timeliness failure is a violation of a time constraint or a violation of sequencing behavior that can be observed externally. That is, testers can actually see the incorrect behavior [6].

There are five practical common problems associated with software testing and particularly in real-time reactive system: Observability, Controllability, Reproducibility, Natural Language Specification, and Coverage Criterion.

Observability: How easy is it to observe the behavior of a program in terms of its outputs, effects on the environment, and other hardware and software components. There are two aspects to Observability. First, we must be able to observe every significant event generated by the environment, and much more important and difficult – determine the

correct ordering and timing of events; secondly, it is necessary to observe the actions and outputs of the system during testing, but without disturbing its timing behavior.

Controllability: How easy is it to provide a program with the needed inputs, in terms of values, operations, and behavior. That is we must be able to control the execution of the system, so that it is possible to reproduce arbitrary test scenarios such that the system's reaction is deterministic.

Reproducibility: Does the system repeatedly exhibit identical behavior when stimulated with the same inputs. Reproducibility is a necessary property for testing, particularly for regression testing and debugging. It is very difficult to achieve reproducibility in real-time systems, especially in event-triggered and dynamically scheduled systems [6, 10].

Natural Language (NL) Specifications: These are descriptions of requirements in a textual format. That is, all timing constraints and temporal behavior are captured as a simple text document. One of the significant problems with NL is that it is rather ambiguous. Ambiguous specifications often will lead to incorrect interpretation and then subsequent errors in system implementation. The following requirement and Figure 5 illustrate an example of ambiguity [9].

Requirement: *Whenever $X > 0$ then $Y > 0$ should follow within 15 seconds afterward.*
The first ambiguity is when do we consider $X > 0$ to occur? Is it when it changes from $X \leq 0$ to $X > 0$, that is time 5 in the Figure 5, or is it every time when $X > 0$ is true. Let's assume the first option and rewrite the requirement.

Modified Requirement: *Whenever $X > 0$ change event occurs, then the $Y > 0$ change should follow within 15 seconds afterward.*

Now, a second ambiguity leads to two new interpretations. The first interpretation considers a scenario when $Y > 0$ happens within 15 seconds of all instances of the $X > 0$ value change event. Given this interpretation, then Figure 5 is correct. However, the second interpretation, considers a scenario when there must be a unique $Y > 0$ change event for every $X > 0$ change event.

In this case Figure 5 is incorrect [9].

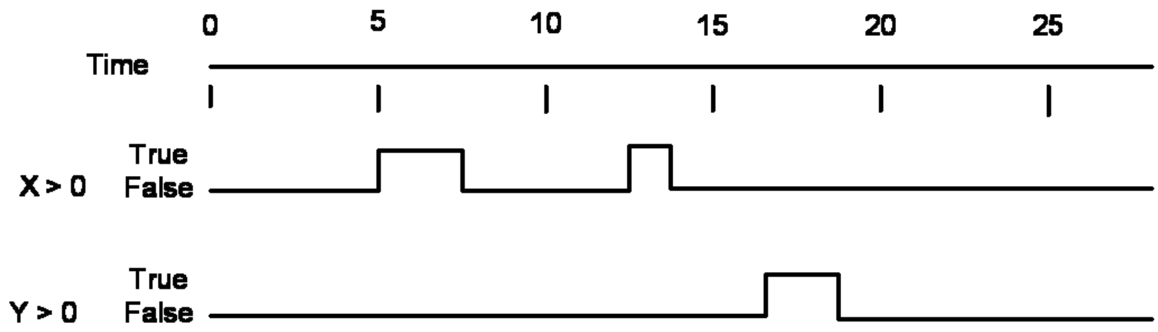

Figure 5: Scenario for ambiguous requirement. From [9].

Coverage Criterion: This refers to the difficulty in finding a suitable test artifact for which test coverage can be examined effectively to test temporal behavior. For example, program code could be modeled as a graph with nodes and edges as shown in Figure 6 such that the nodes could represent the program's methods, and edges between nodes could represent methods calls. From Figure 6, node coverage is a Test Requirement (TR) = {0,1,2} with test path [0,1,2]. Edge coverage is TR = {(0,1), (0,2), (1,2)} with test paths [0,1,2], [0,2]. Recall that coverage criterion is a set of practical rules of when to stop testing. However, it is not readily obvious which criterion, either node coverage or edge coverage, should be used in testing temporal properties (i.e., sequence of events and timing constraints). There is no direct relationship between coverage criterion for a test artifact and temporal behavior. As a result, there are no guidelines for which coverage criterion would be better suited for testing reactive systems.

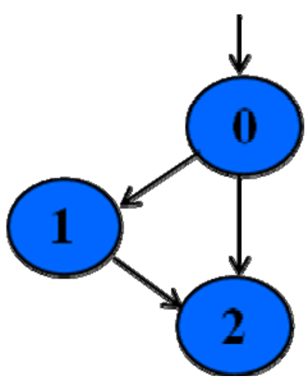

Figure 6: Program modeled as graph

One of the FV&V approaches is theorem proving. It is based on mathematical techniques to make a convincing argument that a program conforms to a formal requirement. For instance, Hoare's triple logic uses pre-condition, post-condition and actual program to be proved. Here is a definition that uses Hoare's triple logic.

Definition of Total Correctness: The triple $(\varphi)P(\psi)$ is satisfied under total correctness if, for all states in which P is executed which satisfy the precondition φ, P is guaranteed to terminate, and the resulting state satisfies the postcondition ψ [12].

Total correctness is based on the assertions, such as precondition and postcondition, expressed as first-order predicate logic. The attempt is to prove that if the precondition holds, then for all possible paths in the given function, the postcondition holds. But unfortunately, it is algorithmically undecidable. There are no means to get a mechanical procedure to prove correctness of any program containing a postcondition written in predicate logic. Predicate logic at its core is reasoning about sets. Predicate logic is very powerful because it can be applied to a wide range of real-life situations. These situations can be formulated using predicate logic, but unfortunately, because of its expressive power almost everything interesting is undecidable. Since total correctness is algorithmically undecidable, there are no automation tools available to fully implement total correctness via static verification, and to prove unequivocally that the entire software program satisfies all the given postconditions. Therefore, theorem proving is mostly a manual task. In addition, total correctness requires proving program termination. Proving that a programming construct such as a loop is going to terminate is found be very challenging and time consuming.

Model checking is another well-known FV&V technique. The inputs to a model checker are a (usually finite-state) description of the system to be analyzed, together with a number of properties, often expressed as formulas of temporal logic [12], that are expected to hold for the system. The model checker either confirms that the properties hold, or reports that they are violated. In the latter case, it provides a counterexample, a run that violates the property. The objective is to convert a given program (e.g., written in C++, Java, or Ada) into a model and then prove correctness of the model. The model should possess a fine balance between size and completeness. Therefore, it should be

small enough to be manageable so that correctness can be proved but also complete enough to capture all the essential behaviors and properties of the program. Kripke structure is used to model system behavior. The Kripke structure will always have a finite number of states, where all transitions between states are possible to implement. That is, each path on the diagram is possible within the behavior of the model. Each path is considered a legal path representing some kind of scenario evolving within the model. Whenever using model checking, the first step is to define a finite state model from the given program where all paths are allowed. However, in contrast to Kripke structure definition, not all paths in real programs (e.g., C++, Java, Ada) are executable. That is, while converting an existing program into a Kripke structure model, there is the possibility of losing certain behaviors that exist in the original source code. This process of converting a program into a model is called abstraction. Not everything converts into a final state transition model, causing certain features to be lost. This means that properties are being verified using the model may not exactly match the intended properties of the original program. In addition, due to the state explosion problem, the abstraction process may reduce the size of program variables. Overall, this presents a serious pitfall for software verification using model checking. For example, a positive verdict from the model checker is then of limited value because errors may well be hidden by the simplifications (abstraction process) that had to be applied to create the model. On the other hand, counterexamples (the model indicating a failure) may be due to modeling artifacts and no longer correspond to actual system runs.

In [3], the authors make a convincing argument that studies of software failures typically point to the importance of correct requirements and the difficulties in getting the correct description of these requirements. That is, a project must start with correct requirements specifications. Otherwise, it does not matter how effective and efficient verification is; it is wasteful to formally verify that a system behaves “correctly” according to invalid requirements [3].

Considering all the difficulties and limitations related to testing real-time reactive systems, the V&V team is faced with real challenges. To address these V&V testing issues, Professor Drusinsky has developed a novel lightweight formal specification, validation, and verification technique.

The technique is novel in two aspects. First, it uses an intuitive, familiar, and diagrammatic notation for formal specification, a notation that is Turing equivalent and supports the capture of real-life requirements. Secondly, the technique includes a computer aided approach for validating the correctness of requirements early in the development process, thus allowing sufficient time for the correction of ambiguous and underspecified requirements. The verification phase is based on off-line verification by using captured log files [4]. Drusinsky has introduced statechart assertions, a new formalism that combines Unified Modeling Language (UML)-based prototyping, UML-based formal specification, runtime monitoring and execution-based model checking [13].

In support of validation and verification testing techniques with this new formalism, Drusinsky has also developed StateRover, a commercially available tool designed for UML statechart design entry, code generation, and visual debugging animation. StateRover is an Eclipse-based tool that fully supports statechart assertion validation and verification.

This thesis attempts to provide a proof of concept demonstration for the V&V process conducted on the real operational software system using the statechart assertion formalism. The real-world application that will be used in this thesis is the Multifunctional Information Distribution System (MIDS) Controller. The primary purpose for the MIDS Controller is to send, receive and process Link-16 tactical data link messages. Link-16 is a primary data link used by joint DoD services (Navy, Air Force, Army and Marines) as well as by NATO for command, control, communication and intelligence. The MIDS Controller runs on a PowerPC single board computer with a VxWorks real-time operating system. This thesis describes an application of rapidly developed UML-based specification, validation, and runtime verification (V&V) to this mission-critical MIDS Controller (MC) software system. This thesis focuses on a

complete V&V process starting with a natural language requirement specification, continuing with formal specification and requirement validation, and ending with verification on a VxWorks-based embedded system target.

III. OVERVIEW OF MIDS CONTROLLER SUT

The EA-6B is the primary tactical electronic warfare aircraft designed to provide lethal and non-lethal electronic support in the suppression of enemy air defenses (SEAD) and Destruction of Enemy Air Defenses (DEAD) in support of Navy, Marine Corps, and Air Force. Capabilities include suppression and degradation of enemy defense systems by airborne electronic jamming and employment of the High Speed Anti-Radiation Missile (HARM). EA-6B key components are electronic surveillance equipment that will rapidly and accurately identify and geo-locate threat systems, and an integrated electronic jamming suite that will effectively jam and degrade threats. The typical EA-6B mission includes detecting, locating, and identifying enemy communication and weapon systems; correlating on-board and off-board data; and employing jamming techniques against those systems. The modern battle space is complex and dynamic, requiring timely and clear information and decisions by all levels of military command. Link-16 supports this requirement by enabling exchange of real-time tactical data among U.S. Navy, Joint Service, and North Atlantic Treaty Organization (NATO) ships and aircraft. Link-16 provides for the rapid and reliable exchange of tactical data at all levels of command, control and operational engagement.

Operationally, Link-16 has the capability of exchanging command and control, navigation, relative positioning, voice and identification data among airborne, ground-based, and shipboard terminals. It consists of a specialized communications network infrastructure operating in the UHF part of the Radio Frequency (RF) spectrum. The Multifunctional Information Distribution System (MIDS) terminal implements Link-16 tactical communication by providing integrated position determination, navigation and present position identification as well as voice and data communication capabilities. With a MIDS terminal, the EA-6B aircraft has a critical advantage in situational awareness by receiving a clear picture of the entire battle space.

Link-16 includes many new features that improve on previous generations of data link communication systems. The new features include: nodelessness, jam resistance, improved security, increased data rate throughput, and reduced data terminal size.

Figure 7 shows a conceptual Link-16 implementation with possible participants.

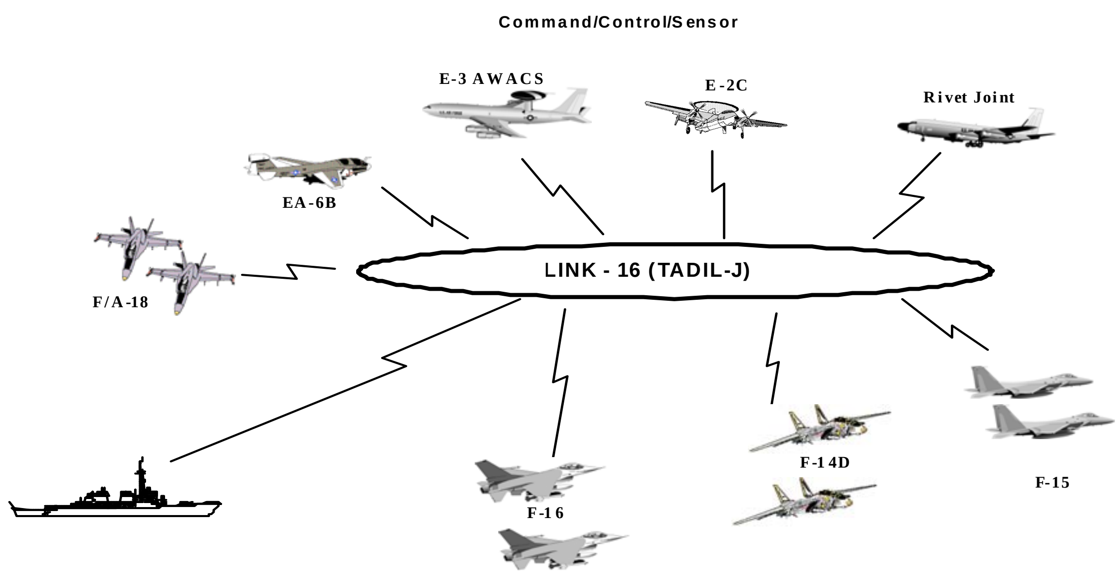

Figure 7: Link-16 participants' implementation. From [14].

In general, MIDS is a hardware communication device that enables Link-16 data and voice communication, as well as access to the Link-16 network.

Any system requiring Link-16 network capabilities has to interface to a MIDS terminal. On the EA-6B aircraft, the MIDS Controller (MC) is a designated host computer to interface with a MIDS terminal.

A. SYSTEM ARCHITECTURE

Figure 8 captures system architecture from the MIDS Controller (MC) perspective:

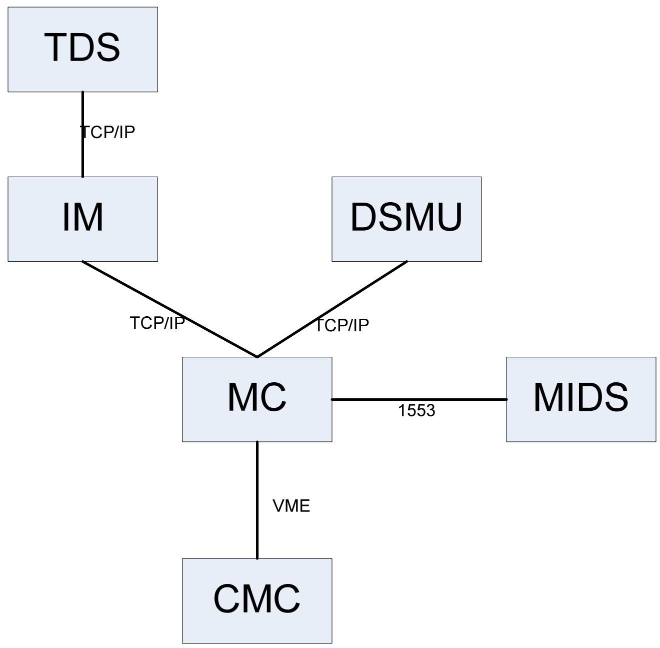

Figure 8: System architecture

1. Subsystems Brief Description

Tactical Display System (TDS): Provides operator controls and displays, Link-16 track management and MIDS terminal control

Information Manager (IM): Manages distributed databases, coordinates Link-16 data between MC and TDS, distributes data to multiple TDSs, mission recording and mission database loading and storage.

Data Storage Memory Unit (DSMU): Stores operational programs, mission database and mission recording data

Multifunctional Information Distribution System (MIDS) Terminal: Link-16 network participation, receive and transmit Link-16 messages. Passes data to and from the MIDS Controller.

Central Mission Computer (CMC): RF jamming management and control of the RF synthesizers and transmitters. Aircraft navigation data control.

MIDS Controller (MC): Intermediary between MIDS terminal and EA-6B integrated distributed systems. Provides data exchange protocol with a MIDS terminal to process Link-16 messages. Implements Link-16 message formatting and message decoding. Maintains all Link-16 active tracks, local and remote track management, and track filtering. In this thesis, MC is considered software under test (SUT) and a target system for applying verification and validation techniques. MC fits the profile of a typical real-time embedded system. It is hosted on a PowerPC single board computer running on the VxWorks real-time operating system.

B. LINK-16 MESSAGES OVERVIEW

The messages exchanged between participating Link-16 platforms are called J-series messages.

Each J-series message is composed of one or more words. All messages have an initial word (I). In addition, they may have one or more extension words (E), and one or more continuation words (C). Each J-series message is mapped to the functional channel (surveillance, air control, Electronic Warfare (EW), etc.,) it supports. These channels are called Network Participation Group (NPGs) and are the building blocks of the network. Various types of operational information are exchanged among users through assigned access to these NPGs.

MIDS Controller (MC) categorizes incoming J-series messages into two broad categories:

Situational Awareness (SA) messages providing data regarding Track/Point entities.

Command messages directing/requesting either the MC or the aircrew to act. Alerts and warnings are included in this category. Some of these messages will be Link-16 Receipt Compliance (R/C) messages requiring an Operator Response message.

C. NL REQUIREMENTS

The NL requirements were taken directly from MC Software Requirement Specification (SRS) document. The NL requirements described in this thesis capture two distinct functional areas: MC Power Up Initialization Sequence and HARM (High Speed Anti-Radiation Missile) Hand Off (HHO).

1. MC Power Up Initialization Sequence

Note that in the NL specifications below, numbered actions must be performed in sequence while subsidiary actions may be performed in any order. Although the order of events within numbered NL sub-requirements 2 and 3 is irrelevant, the MC must first establish 1553 communications with the MIDS terminal and only then get the results of the MIDS terminal BIT, set the LVT terminal state to TDMA only, and obtain the terminal load and net status.

1. The MC shall obtain results of the PowerPC power-on Built-in Test (BIT).

2. The MC will perform the following actions (order is irrelevant):

   a. The MC shall begin transmitting its UDP Broadcast Client Status message-2541.

   b. The MC shall attempt to establish MIL-STD-1553B communications with the MIDS low volume terminal LVT and shall continue to do so until communications are established.

   c. When communications are established, MC shall obtain the results of the Terminal's power-on BIT.

   d. When communications are established, MC shall set the LVT Terminal State to Time Division Multiple Access (TDMA) Only.

   e. When communications are established, MC shall obtain the terminal's current load and net entry states.

   f. The MC shall attempt to establish a TCP connection with the IM processor.

g. The MC shall set up an VME bus interrupt capability to receive an external interrupt from CMC.

3. Once the TCP socket is established, MC will send the following messages to the TDS (order is irrelevant):

a. MC shall send the On-Demand BIT (ODB)/Power-On BIT Results message-2544, containing results of PowerPC and MIDS LVT power-on BIT.

b. MC shall send the MC Software Version message-2547.

c. MC shall send the Link-16 Network Status message-2545.

MC shall complete entire initialization sequence within 10 seconds.

2. HHO Requirements

The High Speed Anti-Radiation Missile (HARM) is used against detected radar sites. Link-16 is used to coordinate HARM missile launches between platforms with surveillance capabilities and HARM shooters. For example, the EA-6B has a sophisticated surveillance system, but might not have a HARM missile on board. F-16 has HARM launching capabilities but does not have a sophisticated surveillance detection system. In this case, Link-16 uses a series of J-Messages, J3.5 (radar location), J12.6 (HARM parameters), and J12.0 (actual command to shoot HARM missile) to coordinate HARM shots between EA-6B and F-16.

HHO Requirements are:

- Upon receiving from IM message-2572 (Local Track Report) with HHO field set to true, MC shall: create a local track in the database, assign loopbackID to a track number, set retry count to one, and transmit a J3.5 (Land Point) message to MIDS terminal.
- MC shall receive loopbackID response message from the MIDS terminal for the J3.5 message. If the loopbackID from MIDS indicates transmit status failure, MC shall re-transmit the J3.5 message once. If the

loopbackID from MIDS indicates transmit status failure from the re-transmit J3.5 message, MC shall abort HHO processing.

- Upon receiving the loopbackID from MIDS indicating transmit success for the J3.5 message, MC shall assign a loopbackID to a track number, setup retry count to one and transmit J12.6 (HARM DA parameters) message to MIDS terminal.

- MC shall receive the loopbackID response message from the MIDS terminal for the J12.6 message. If the loopbackID from MIDS indicates transmit status failure, MC shall re-transmit the J12.6 message once. If the loopbackID from MIDS indicates transmit status failure from re-transmit J12.6 message, MC shall abort HHO processing.

- Upon receiving loopbackID from MIDS indicating transmit success for the J12.6 message, MC shall wait for the IM to send to MC Message-5362 (Mission Assignment).

- Upon receiving from IM message-5362 (Mission Assignment), and only if MC had already successfully processed J3.5 and J12.6, MC shall assign loopback to a track number, setup retry count to one and transmit J12.0 Mission Assignment.

- MC shall process message-5362 from IM only if MC had successfully sent J3.5 and J12.6 messages.

- In case IM sends Message-5362 prior to Message-2572, MC must wait for Message-2572 from IM, in order to transmit the J3.5 (Land Point) and J12.6 (HARM DA parameters) messages prior to transmit J12.0 (Mission Assignment).

- MC shall receive the loopbackID response message from the MIDS terminal for the J12.0 message. If loopbackID from MIDS indicating transmit status failure, MC shall re-transmit J12.0 message once. If

loopbackID from MIDS indicating transmit status failure from re-transmit J12.0 message, MC shall abort HHO processing.

- Upon receiving the loopbackID from MIDS indicating transmit success for the J12.0 message, MC shall wait for the J12.0 message from the MIDS terminal with a Receipt Compliance fields indicating the following response: either Will Comply or Cannot Comply status.
- Upon receiving J12.0 message from MIDS terminal with Cannot Comply response, MC shall remove local track from the database.

IV. SPECIFICATION AND VALIDATION PHASE

This phase focuses on capturing and expressing NL requirements as statechart assertion diagrams specification and the validation of the statechart assertions.

A. STATECHART ASSERTIONS

The challenge in validating NL requirements that describe temporal sequencing behavior is that, currently, computers cannot validate NL specifications. That is, computers can parse NL text, but do not really know the actual intent and meaning of the specification. Certainly, there is a need for some validation technique as well as a formal notation language in expressing NL requirements, which would facilitate computer assisted validation. Many literature sources have suggested using finite state machine and their corresponding state diagrams as a formal mechanism to expressing dynamics of the system [11]. State diagrams are simply directed graphs, with nodes denoting states, and arrows denoting transitions. However, early in their use, it became apparent that complex systems could not be beneficially expressed using state diagrams, because of an unmanageable, exponentially growing number of states. This resulted in unstructured, difficult to comprehend, chaotic state diagrams [11]. Harel proposed the use of statechart diagrams as a visual approach to modeling the behavior of complex reactive systems [11]. Statecharts are now considered a standard part of UML. Specifically, statecharts constitute a visual formalism for describing states and transitions in a modular fashion, enabling clustering (i.e., superstate), orthogonality (i.e., concurrency) and refinement [11]. For example, consider the given finite state machine diagram as illustrated in Figure 9.

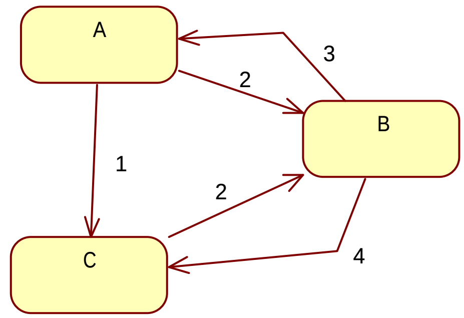

Figure 9: Finite state machine diagram. From [11].

This diagram could be expressed as a statechart diagram shown in Figure 10.

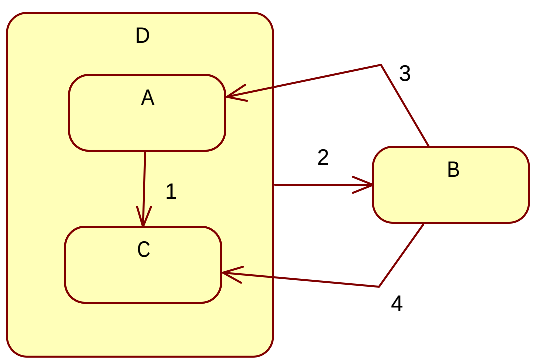

Figure 10: Statechart superstate. From [11].

Notably, since event 2 makes a transition to B from either A or C we can use the super state D and cluster A and C as sub states of D. The semantics of D is then exclusive-or (XOR) of A and C. That is, to be in state D, it is valid to be in either A or B state, but not both [11].

Harel [11] demonstrates an example of orthogonality as shown in Figure 11.

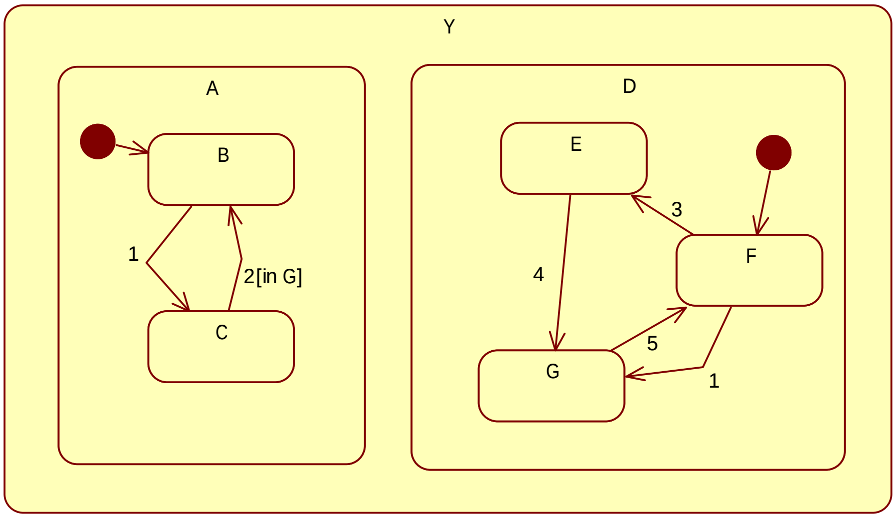

Figure 11. Orthogonality example. From [11].

In this statechart, event 1 would cause a simultaneous transfer B to C and F to G. Also, event 2 is a conditional event and can take place only if system is in G state. The system starts simultaneously from B and F states as illustrated by the red dot in the diagram.

Figure 12 now captures the same behavior in a state diagram

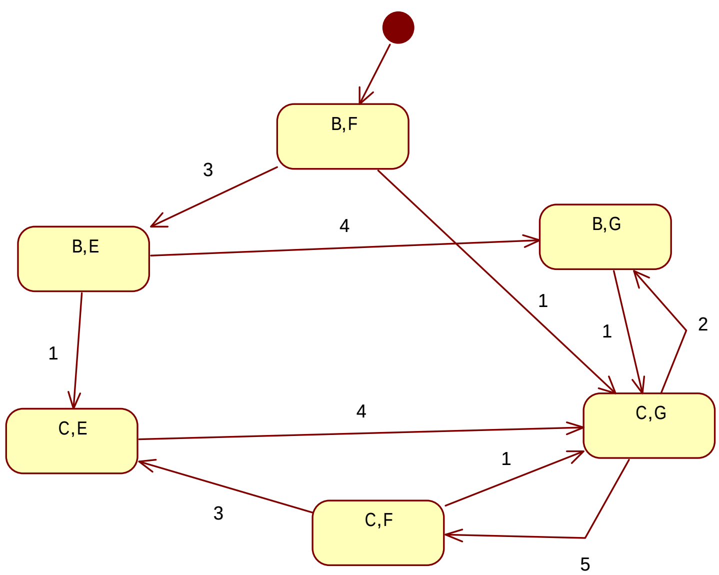

Figure 12: State diagram without statechart orthogonality. From [11].

The resulting state diagram has six states, which came from the product (multiplication) of the two possible states, C and D, in the one component, and the three possible states E, F and G, in the other component. Obviously, 100 states in each component would result in a diagram with 10,000 states. This example clearly illustrates possible state explosion with a classical state diagram.

Drusinsky [15] extended the use of statechart diagrams into statechart assertion diagrams in two ways to specify formal assertions:

1. It includes a built-in Boolean flag bSuccess (and a corresponding isSuccess method) that specifies the Boolean status of the assertion (true if the assertion succeeds and false if the assertion fails) [15].

2. Assertion statechart to be nondeterministic [15].

Here, the novel approach in statechart assertion diagrams is that any reactive system with a complex temporal behavior can be expressed as a statechart diagram. The statechart assertion diagram can further be converted to a software program (e.g., developed in Java). This allows computer assisted validation to be done on the software program, which captured the original specifications. The big difference between NL text and statechart assertion is that a computer does not understand NL text, but it does understand statechart assertion diagrams, which, as a result, are being converted into a software program (assertion repository).

A statechart assertion has two basic components, states and transitions.

States represent system memory with attributes and their values. Changing states causes certain attributes to change their values. States can have potential actions. A state action can be executed when the state is entered or exited. Statechart *transitions* are annotated with events, conditions and actions. The syntax for a statechart transition is Event[condition]/action, as shown in Figure 13 [9].

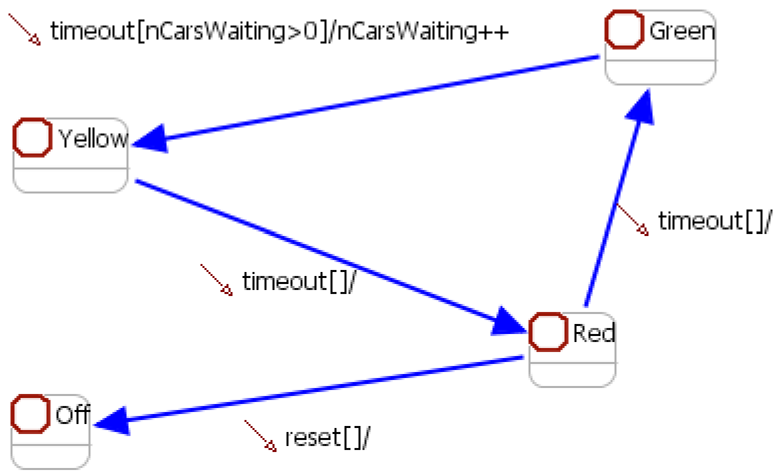

Figure 13: General syntax of a statechart transition. From [9].

As an example, Figure 14 illustrates a statechart assertion for the following requirement. R1: *Missile launch must not occur, when system is in jamming radiate mode.*

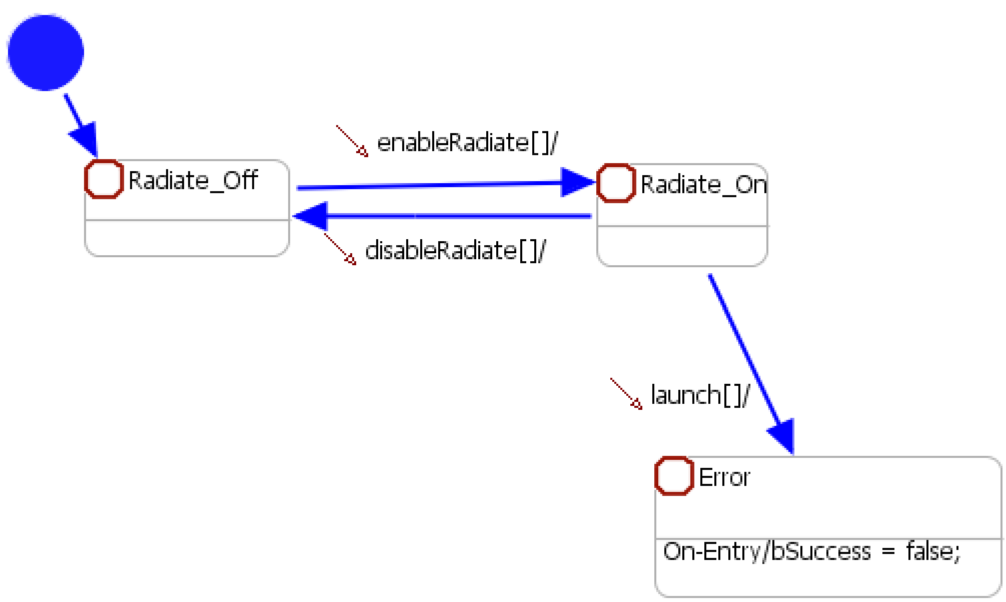

Figure 14: Statechart assertion diagram for R1 requirement

The statechart diagram shows a state transition from Radiate_On to Error state upon receiving a launch event while in Radiate_On state. In Error state, bSuccess is set to false. This simple example captures the essence of the statechart assertions. Undesired or illegal events are being expressed via transitioning to an Error state.

Requirements concerned with real-time constraints are demonstrated with the following example. Consider the following requirement.

R2: *The system must establish network connection within one second of being turned on.* Figure 15 illustrates a statechart assertion for the R2 requirement.

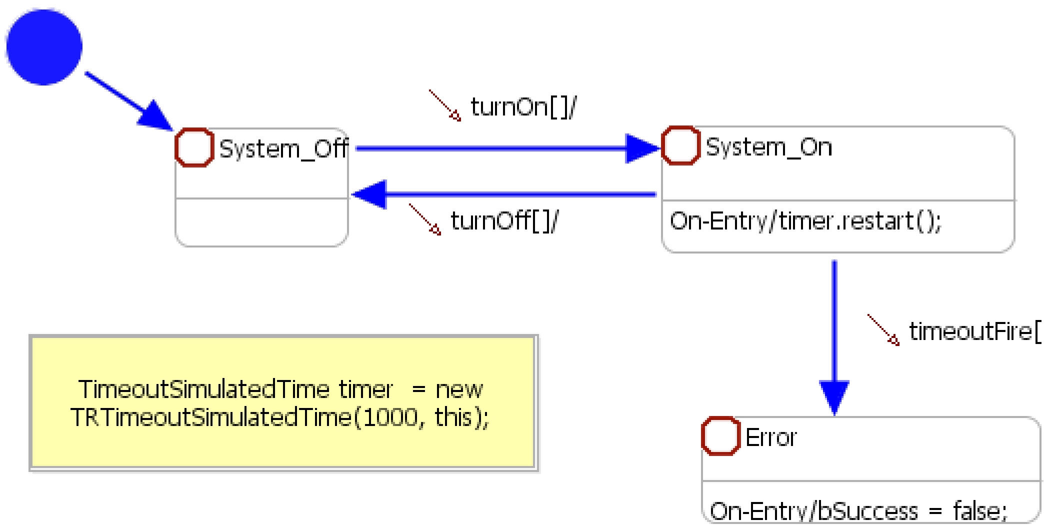

Figure 15: Statechart assertion diagram for R2 requirement

In the statechart assertion diagram, the variable timer is being declared and initialized to a 1000-millisecond time constraint. Upon entering System_On state, the timer restarts the timer countdown from 1000 milliseconds. The timeoutFire event occurs when the timer count reaches 0, causing transition to Error state. This statechart assertion diagram is capturing a time constraint violation.

Another example, from [9], illustrates the use of default events. Consider the following NL requirement.

R3: *No event other than P is allowed between an event Q and a subsequent event R.* Figure 16 captures the statechart assertion for R3 requirement.

This example shows how to assert specific order within a sequence of events. Here, once state A has been reached due to Q event, no other event is valid, except P or R. Thus, while in state A, receiving Q event would trigger eventTRDefault causing transition to Error state. The Default event resembles an else condition from an if-then-else conditional block of code [9].

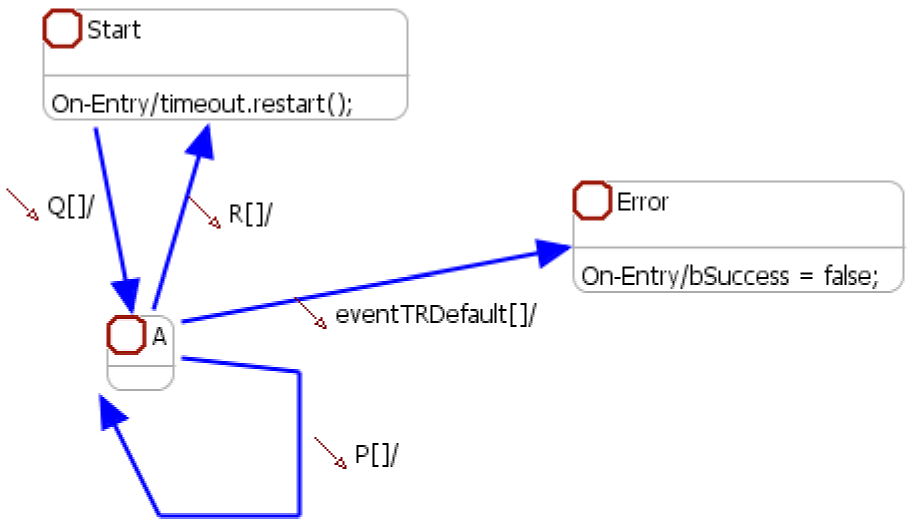

Figure 16: Statechart assertion diagram for R3 requirement. From [9].

The statechart assertions for two functional areas, MC Power Up Initialization Sequence and HHO, were developed and validated inside the special container called assertion repository. This is a special Eclipse project that provides the environment for subsequent verification described in verification phase section. Figure 17 depicts the Eclipse view of the assertion repository for the MC controller [16]. In addition, the repository also provides a main driver for collectively executing all requirement assertions. JUnit tests created from log files fire log-file events and data assignments through this driver, which subsequently dispatches them to all assertions [4].

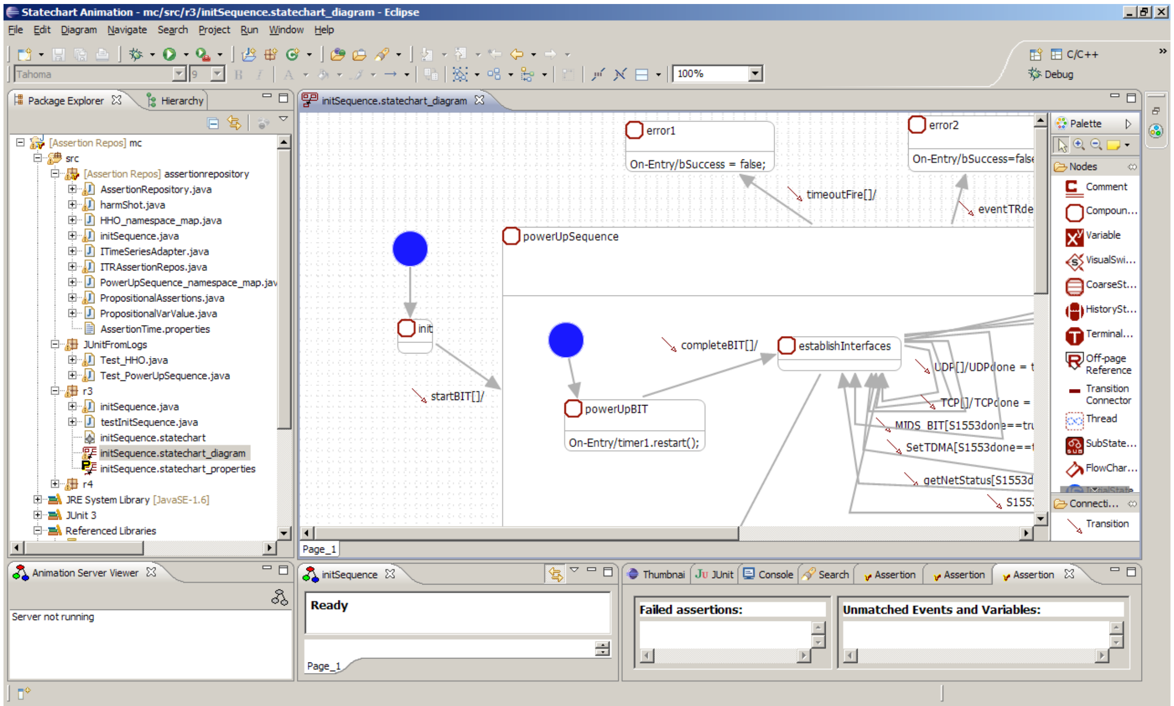

Figure 17: MC Validation and Verification with assertion repository

B. VALIDATING MC POWER UP INITIALIZATION SEQUENCE

In validating the MC power up sequence requirements described in section III.C.1, the first step is to capture NL requirements in a statechart assertion diagram. The MC power sequence expresses a specific order of events, and a time constraint of 10 seconds. However, the order within lettered sections is not important. For example, MC power up BIT (Built-In-Test) (section 1) must occur prior to establishing external communication (section 2). But events such as establishing TCP connections and setting up VME interrupts (lettered steps), could be in any order. By analyzing MC power up sequence requirements, it becomes apparent that these requirements are somewhat under specified. For example, it is clear that power up BIT should occur prior to establishing external communication. However, is it legal to have the power up BIT event occurring again while establishing external communications? What if the power up BIT event never occurs; is it still legal to establish external communications?

Also, MC power up sequence requirements assume that all external devices are already powered up. For example, if MIDS terminal is not available while MC powers

up, as a result, MC would fail to establish 1553 communication with the MIDS terminal and would not acquire Link-16 network status. In this case, does MC still have to send Link-16 Network Status message-2545 even though MC does not have any network status information? These are valid questions, which arise naturally while attempting to express NL requirements in a statechart assertion diagram. For the purposes of this thesis, the assumption is made that once the power up sequence has started, all single valid events need to occur in the sequential order. For example, Link-16 network status cannot be sent until MC establishes 1553 communication with a MIDS Terminal and acquires Link-16 network status. Figure 18 depicts the statechart assertion diagram for the MC power up initialization sequence. It captures the following NL concerns cited in Section III.C.1:

1. The NL requirement specifies a strict sequence order for sub-requirements 1, 2, and 3.

2. There exists a 10-second upper bound time constraint for the entire initialization sequence.

3. The order of events within numbered NL the requirements related to Interface Initialization (Step 2) and Send Messages (Step 3) is irrelevant, except for MC having to first establish 1553 communication with a MIDS terminal and only then get MIDS terminal BIT, set TDMA, and get terminal load and Link-16 network status.

Since MC power up sequence requirements are interdependent on each other, it is more intuitive to capture these requirements with one single assertion statechart diagram.

This statechart assertion diagram uses super state and sub-state hierarchy to reduce diagrammatic complexity already discussed in section IV.A. Specifically, any events originating from a super state are all applicable to the sub-states and implied to be originating from any of the sub-states. For example, eventTRDefault from powerUpSequence state is applicable to powerUpBIT, establishInterfaces, and sendInitMessages states. An out of order event in any of these states would cause default event eventTRDefault transition to Error2 state.

A statechart assertion diagram specifically enforces assertions dealing with time constraint and order of events. A given time constraint is enforced by initializing a timer1 to 10 seconds in the powerUpBIT state, and then event timeoutFire would transition to the Error1 state whenever the event occurs of timer1 reaching its 10-second timeout interval. The order of events is enforced by using eventTRDefault. For example, by receiving completeBIT event while in the establishedInterfaces state would cause eventTRDefault transition to Error2 state.

Another example, by receiving completeInterfaces event while in the powerUpBIT state would cause eventTRDefault transition to Error2 state. In order to assert that order is not important within Interface Initialization requirement, all the events originating in the establishInterfaces state (except completeInterfaces) are looped back to establishInterface state. Statechart assertion diagrams have the capability and syntax to define and initialize variables that could be used for timing and conditional events. The variables are shown in the separate shaded box. There are three conditional events depending on already established 1553 interface. Conditional events such as: MIDS_BIT, SetTDMA and getNetStatus will occur only when Boolean variable S1553done is set to true. Event S1553 sets Boolean variable S1553done to true as an action of its occurrence. Likewise, within sendInitMessages state, events such as softwareVersionMsg, BITResultsMsg, and L16NetworkStatus are looped back to sendInitMessages to assert order independence. Conditional event completeInterfaces depends on all the Boolean variables set to true in the establishInterfaces state. This asserts the order where completeInterfaces event can only occur after all events from establishInterfaces have occurred.

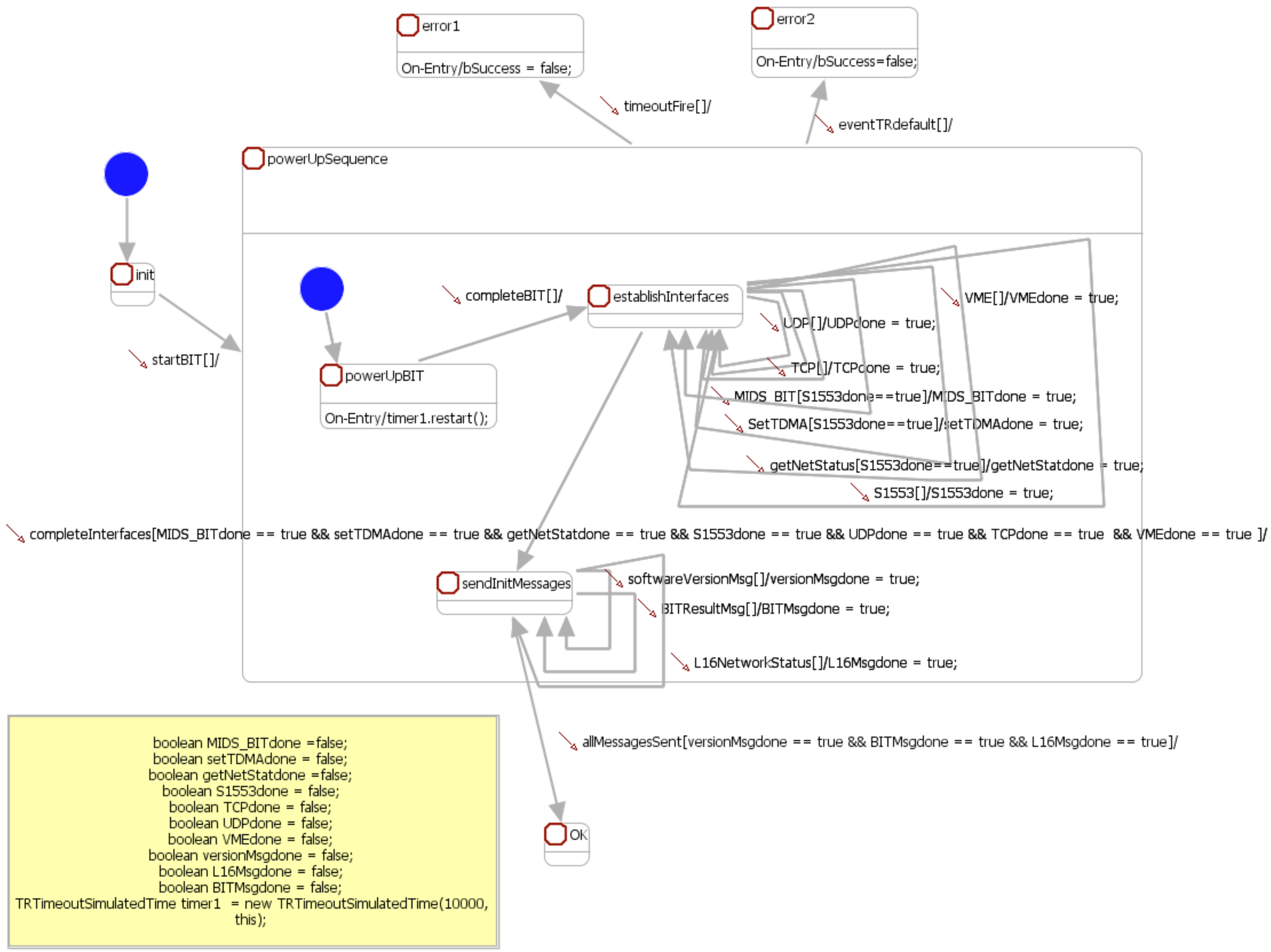

Figure 18: Statechart assertion diagram for MC Power Up Initialization Sequence

The StateRover tool automatically generates Java code from statechart assertions diagrams in an assertion repository. At this stage, all NL requirements have been expressed as executable Java code. This executable code is used by JUnit to validate statechart assertions.

After statechart assertions are completed, the next step is to run validation on statechart assertions. This is one of the key elements in the requirements validation, and it is used to ensure that the statechart assertion correctly represents the intended behavior. Validation testing is accomplished using the JUnit test framework and user-developed test cases. Validation testing is applied to the generated Java code assertion repository. The test cases are easily developed as Java code to validate the statechart assertion diagrams. The JUnit test framework serves as a test driver, invoking user-developed test cases against the assertion repository. Listing 2 illustrates a complete validation test case against the MC power up initialization sequence.

```
public void testRun1()
//all events comply with a specified order of events
{
   sequence.startBIT();
   sequence.incrTime(1000);
   sequence.completeBIT();
   sequence.TCP();
   sequence.UDP();
   sequence.VME();
   sequence.S1553();
   sequence.MIDS_BIT();
   sequence.SetTDMA();
   sequence.getNetStatus();
   sequence.completeInterfaces();
   sequence.L16NetworkStatus();
   sequence.BITResultMsg();
   sequence.softwareVersionMsg();
   sequence.allMessagesSent();
   assertTrue(sequence.isSuccess());
}
```

Listing 2: Validation test

Typically, at the end of each test case, there is an assert statement, which could be assertTrue or assertFalse. Notice, the sequence.isSuccess function returns the value of the Boolean bSuccess variable. Assert statements assertTrue or assertFalse simply assert that the predicate is true or false. The test assertTrue(true) will pass the test. On the other

hand, assertTrue(false) will not pass the test. Similarly, assertFalse(false) will pass the test, and assertFalse(true) will not pass the test. For example, test case 1 has all the events in expected order according to NL requirements and statechart assertion diagram, and as a result, assertTrue is being used at the end of the test case. On the other hand, Listing 3 illustrates a test case capturing a timing constraint violation.

```
public void testRun2()
{
  sequence.startBIT();
  sequence.incrTime(1000);
  sequence.completeBIT();
  sequence.TCP();
  sequence.UDP();
  sequence.S1553();
  sequence.MIDS_BIT();
  sequence.SetTDMA();
  sequence.getNetStatus();
  sequence.completeInterfaces();
  //forces a timeout be exceeding required time limit
  sequence.incrTime(10000);
  sequence.L16NetworkStatus();
  sequence.BITResultMsg();
  sequence.softwareVersionMsg();
  sequence.allMessagesSent();
  assertFalse(sequence.isSuccess());
}
```

Listing 3: Test case for time constraint violations

Test case 2 uses incrTime function to set up system time. Specifically, the function incrTime is cumulatively adding time expressed in milliseconds. For example, the first call sequence.incrTime(1000) increments time to 1000 milliseconds. The second call, sequence.incrTime(10000) increments time to 11000 milliseconds, or 11 seconds. The increase to 11 seconds causes event timeoutFire transition to Error1 state. Therefore, assertFalse is being used at the end of the use case. If assertTrue were used instead of assertFalse, this test case would fail the test.

Figure 19 shows the successful validation test results for all seven test cases.

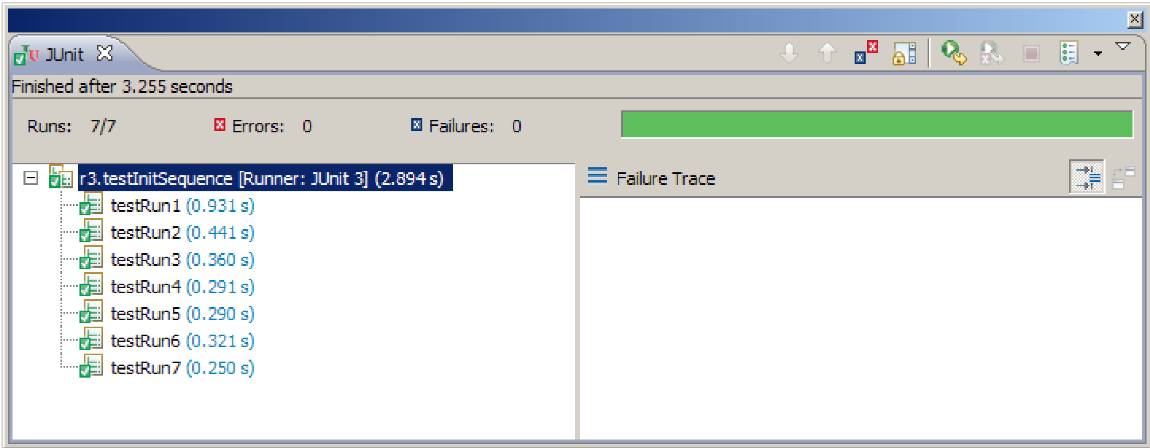

Figure 19: Validation test results (all test cases passing)

Test case 4, illustrated in Listing 4, shows an example for out of order event sequencing. Clearly, MIDS_BIT event cannot occur prior to S1553 event, which is responsible for establishing 1553 interface with a MIDS terminal. Since this case violates order of events, assertFalse is used at the end of the test case. If assertTrue were used instead of assertFalse, test case 4 would fail validation test as shown in Figure 20.

```
public void testRun4()
{
  sequence.startBIT();
  sequence.incrTime(1000);
  sequence.completeBIT();
//MIDS BIT can't be run prior to establishing 1553 interface (event
//S1553)
  sequence.MIDS_BIT()
  sequence.S1553();
  sequence.TCP();
  sequence.UDP();
  sequence.SetTDMA();
  sequence.getNetStatus();
  sequence.completeInterfaces();
  sequence.incrTime(5000);
  sequence.softwareVersionMsg();
  sequence.L16NetworkStatus();
  sequence.BITResultMsg();
  sequence.allMessagesSent();
  assertFalse(sequence.isSuccess());
}
```

Listing 4: Test case for out of order sequencing violation

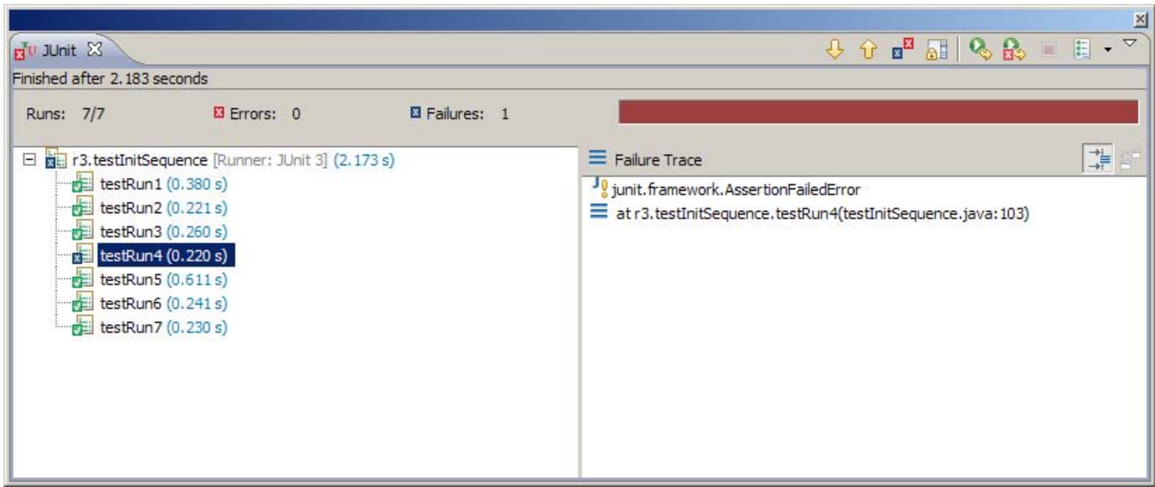

Figure 20: Validation test results (test cases 4 failed)

One of the very helpful feature of StateRover is to capture and display timeline diagrams. A timeline diagram captures events and their associated time stamp of occurrence. The timeline diagrams of Figure 21 and Figure 22 depict success and failure cases, respectively. Specifically, Figure 21 depicts a test case for a scenario which conforms to the NL, whereas Figure 22 depicts a test case for a scenario which violates the NL. Timeline diagrams are helpful in visualizing occurrences of all events with respect to time.

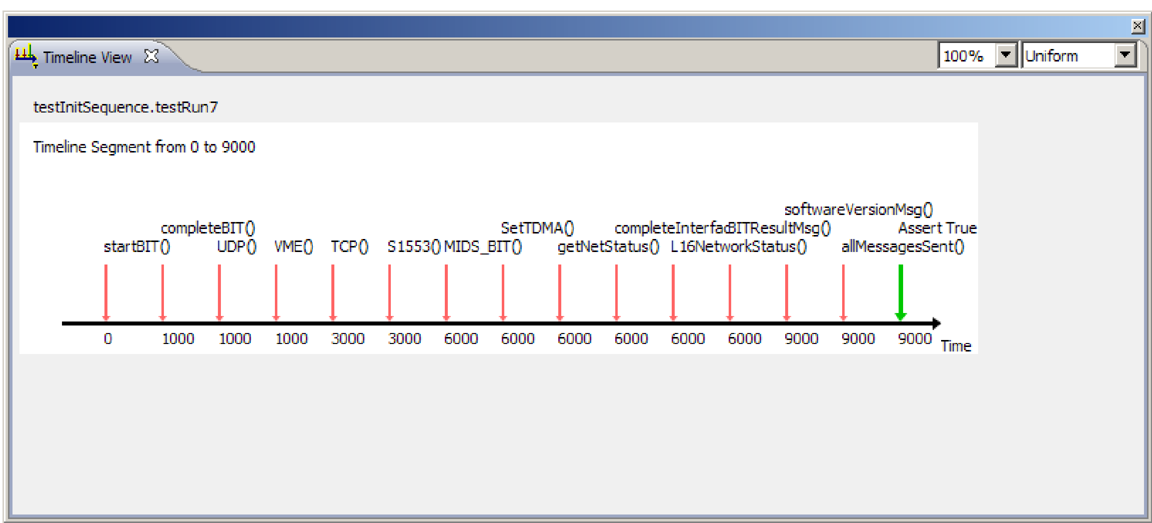

Figure 21: Timeline diagram rendering of a validation test expecting a success. From [16].

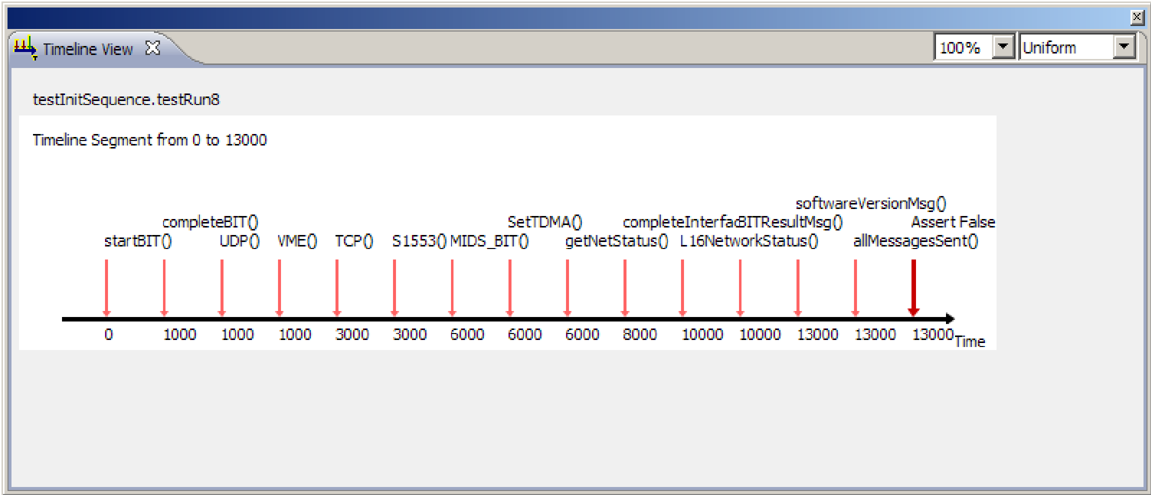

Figure 22: Timeline diagram rendering of a validation test expecting a failure. From [16].

Another helpful feature of StateRover is state animation. State animation works in a few modes: regular and cumulative. In a regular mode, it captures and displays events leading to the current state and the previous state. In a cumulative mode, animation captures and displays all the events and state that had occurred during test run time.

Figure 23 illustrates an example of animation capturing test case 1. This animation diagram highlights all the events and states that have occurred. In this example, test case 1 has assertTrue conforming to NL requirement. Thus, neither Error1 state nor Error2 state is highlighted. Figure 24 illustrates an example of animation capturing test case 2. In this example, test case 2 has assertFalse violating an NL requirement. Thus, Error1 is highlighted indicating a time constraint violation. Logically, animation not only is showing events and states that have occurred, but also indicates the states and events that have not occurred. In this example, upon the expiration of the 10-second time constraint, timeoutFire event transitioned to an Error1 state, thus preventing other events (e.g., MIDS_BIT, SetTDMA) from executing, or other states from being reached (e.g., SendInitMessages, OK).

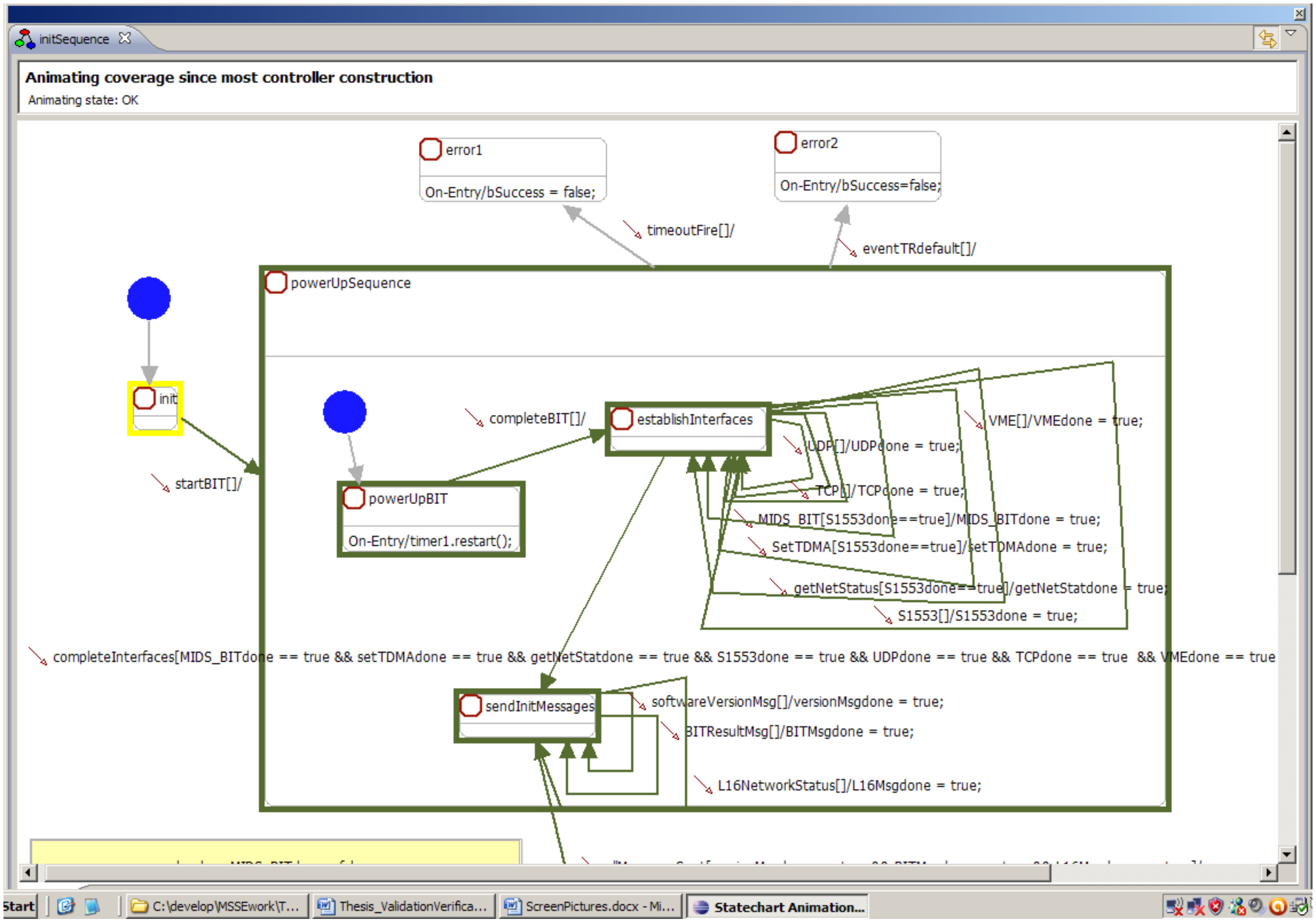

Figure 23: Animation for test case 1

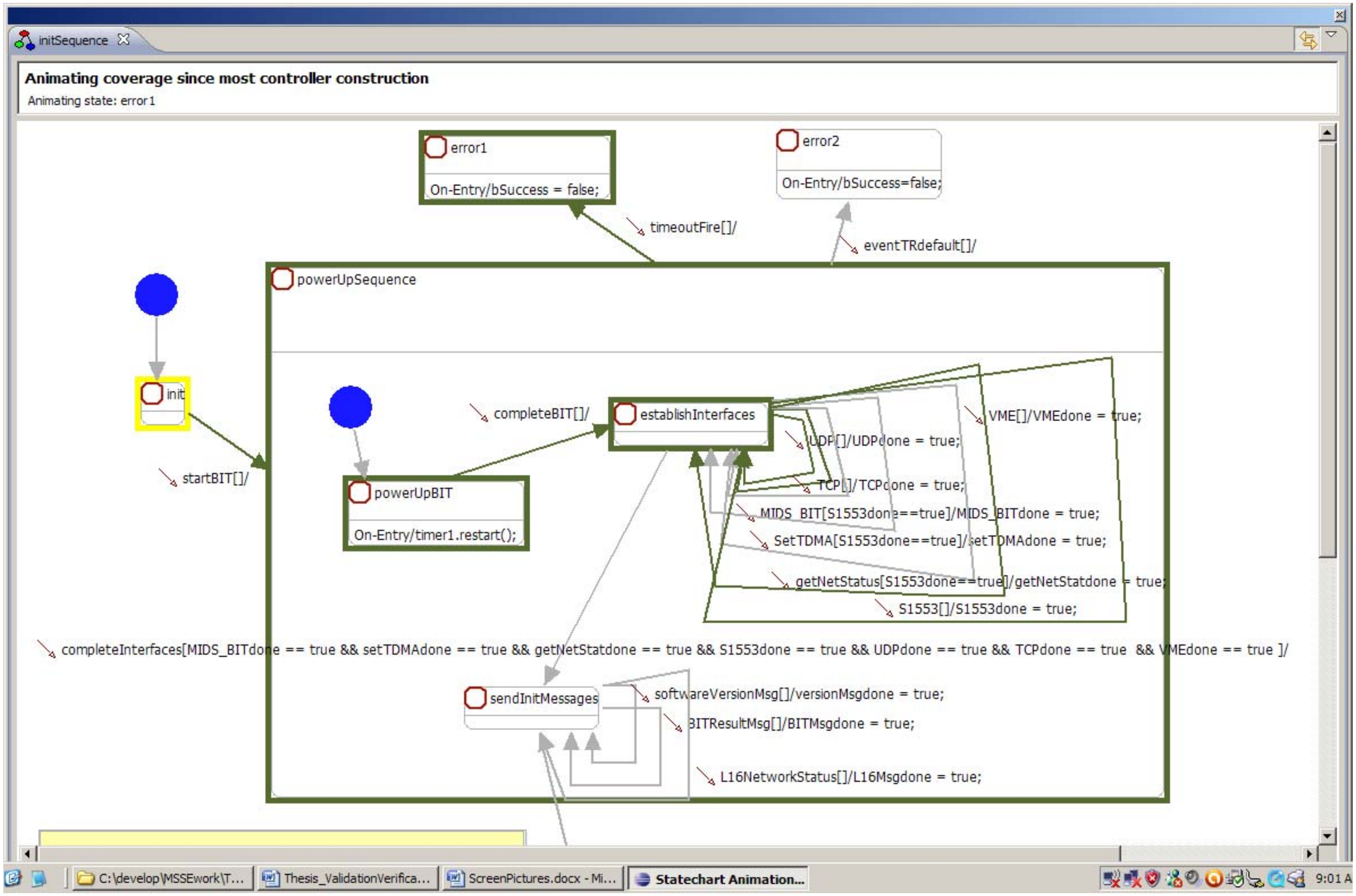

Figure 24: Animation for the test case 2

C. VALIDATING HHO (HARM HAND OFF)

Similar to MC Power Up Initialization, in validating HHO requirements described in Chapter III.C.2, the first step is to capture NL requirements in a statechart assertion diagram.

HHO requirements express a specific order of events, but they do not have any time constraints. Analysis of HHO requirements has led to the realization that NL requirements are somewhat under-specified. For example, what happens when the loopback for the J3.5 message is never received? Does the system have to wait indefinitely for the J3.5 loopback? While it is waiting for the J3.5 loopback, is it valid to receive another Msg-2572 to start another HHO sequence? Once again, these are valid questions, which arise naturally while attempting to express NL requirements in a statechart assertion diagram.

Typically, these questions should be resolved between all involved parties, customer, V&V team and developers. The key point is that statechart assertion diagrams bring out any NL fallacies by making temporal behavior scenarios visible, which provides an opportunity to detect any loopholes within the specification.

For the purposes of this thesis, the assumption was made that once the HHO sequence has started, all single valid events must occur in the sequential order specified in Section III.C.2. There are no timeouts and the system waits indefinitely for any loopback confirmations. Figure 25 depicts the statechart assertion diagram for HHO. The statechart assertion diagram expresses the following main constraints cited in Section III.C.2:

1. All J-message transmissions must occur only after receiving Msg-2375 from IM.
2. Subsequent to the sending of each J-message (J3.5, J12.6, J12.0), the system must wait for the respective loopback confirmation status from MIDS terminal.
3. J-messages must be sent in a particular order: J3.5, J12.6, J12.0.
4. Message J12.0 must be sent only after receiving Msg-5362 from IM.

Since HHO requirements are interdependent between each other, it is more intuitive to capture these requirements with one single assertion statechart diagram.

An HHO statechart assertion diagram consists of three super states: J3_5_sequence, J12_6_sequence and J12_0_sequence. Each super state contains sub-states asserting a loopback confirmation message from the MIDS terminal. The Boolean visual switch within the diagram assists in routing logical flow of events. For example, "J3_5_fail == true" switch depending on the true or false condition directs event flow to another visual switch (J3_5_cnt < 1) or setup_J12_6 state respectively. The diagram asserts one retransmit attempt in case of failed feedback message from MIDS terminal. This is facilitated by using variable J3_5_cnt. Notice that send_msg_J3_5 event from wait_loopback_J3_5 state is incrementing J3_5_cnt as an event action. Analogous to J3_5_sequence super states, J12_6_sequence and J12_0_sequence use exactly the same approach for loopback message confirmation from MIDS terminal, with the exception of using different Boolean visual switch variables and message counters. In addition, J12_6_sequence implements assertion for J12.0 message, which can only be sent after receiving Msg-5362. This assertion is accomplished by wait_msg_5362 state. Listing 5 show a validation test case 1.

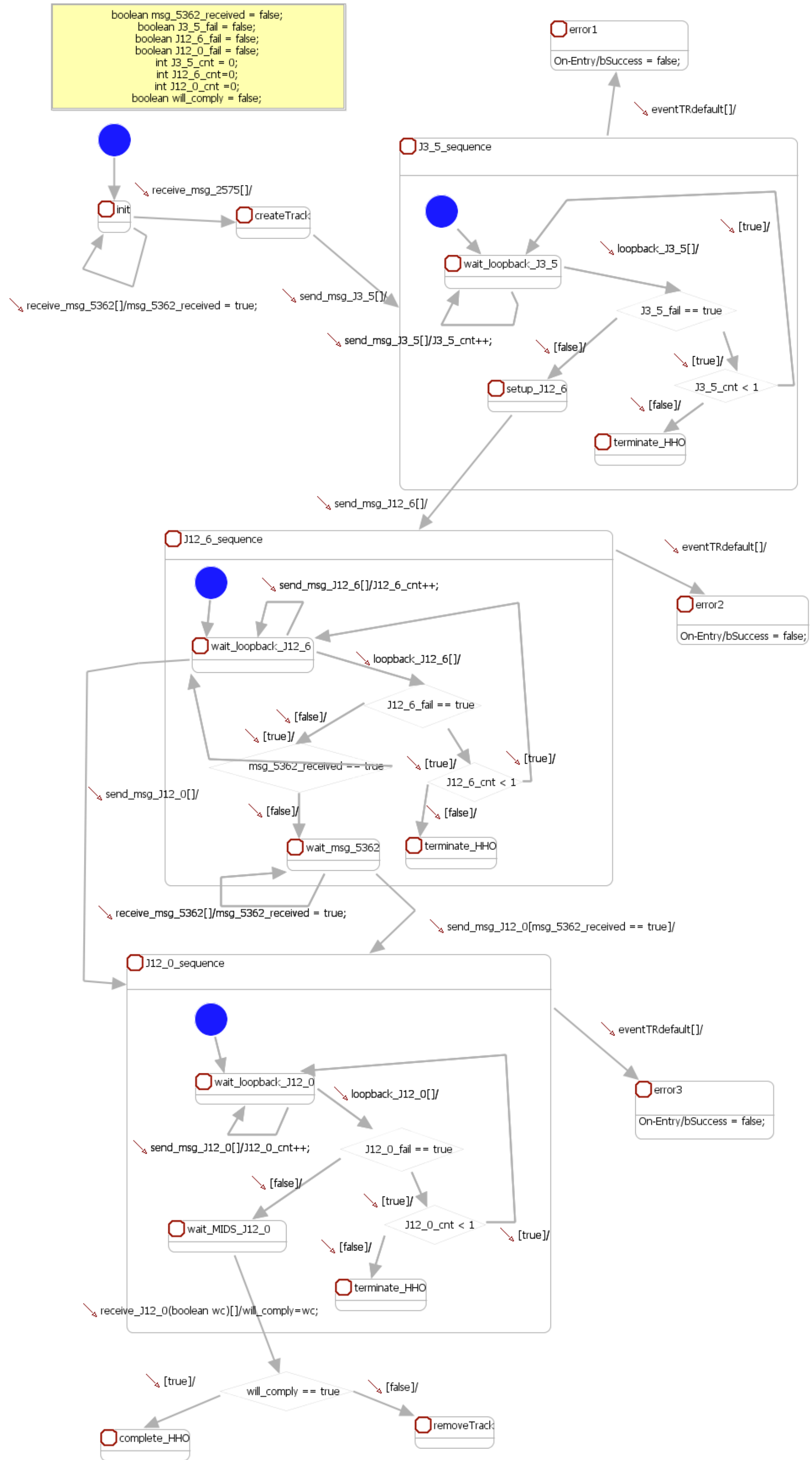

Figure 25: Statechart assertion diagram for HHO

Listings 5, 6 and 7 illustrate complete validation test cases against the HHO statechart assertion diagram.

Test case 1 uses assertTrue indicating an expected order of events. Test case 2 validates a failure of receiving J3.5 loopback confirmation. Consequently, test case 2 uses assertFalse. Test case 3 validates an expected order of event with J3.5 message exercising a second attempt due to a loopback confirmation failure on the first attempt.

```
public void testRun1()
{
   shot.receive_msg_2575();
   shot.send_msg_J3_5();
   shot.loopback_J3_5();
   shot.send_msg_J12_6();
   shot.loopback_J12_6();
   shot.receive_msg_5362();
   shot.send_msg_J12_0();
   shot.loopback_J12_0();
   shot.will_comply = true;
   shot.receive_J12_0(wc);
   assertTrue(shot.isSuccess());
}
```

Listing 5: Validation test case 1

```
public void testRun2()
{
   shot.receive_msg_2575();
   shot.send_msg_J3_5();
   shot.J3_5_fail = true;
   shot.loopback_J3_5();
   shot.send_msg_J3_5();
   shot.loopback_J3_5();
   shot.send_msg_J12_6();
  //never got successful J3.5 loopback confirmation, and yet
//proceeding with J12.6
   shot.loopback_J12_6();
   shot.receive_msg_5362();
   shot.send_msg_J12_0();
   shot.loopback_J12_0();
   shot.will_comply = true;
   shot.receive_J12_0(wc);
   assertFalse(shot.isSuccess());
}
```

Listing 6: Validation test case 2

```java
public void testRun3()
{
   shot.receive_msg_2575();
   shot.send_msg_J3_5();
   shot.J3_5_fail = true;
   shot.loopback_J3_5();
   shot.send_msg_J3_5();
   shot.J3_5_fail = false; //second attempt to sent J3.5 was successful
   shot.loopback_J3_5();
   shot.send_msg_J12_6();
   shot.loopback_J12_6();
   shot.receive_msg_5362();
   shot.send_msg_J12_0();
   shot.loopback_J12_0();
   shot.will_comply = true;
   shot.receive_J12_0(wc);
   assertTrue(shot.isSuccess());
}
```

Listing 7: Validation test case 3

Figure 26 illustrates a timeline diagram for test case 2. This test case did not use incrTime function, thus the time remains zero for all the events. The timeline diagram also captures instances whenever any variable is being set. In this case, two variables J3_5_fail and will_comply, are being set.

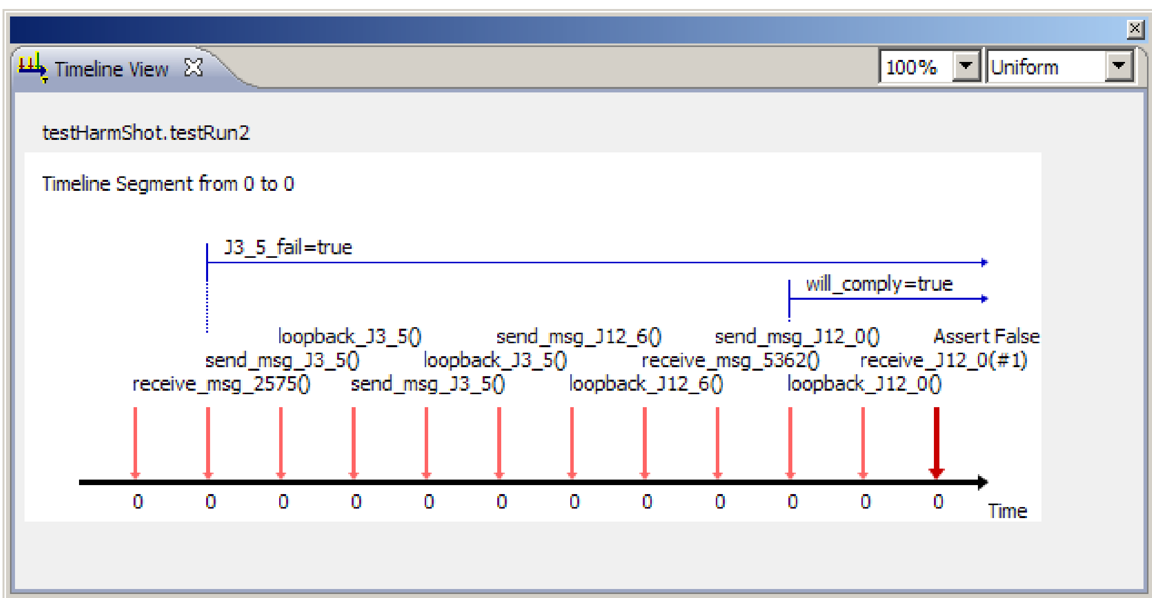

Figure 26: Timeline diagram for test case 1

Typically, there are two kinds of errors in statechart assertions:

1. Implementation errors resulting from mistakes in the statechart assertion [15].

2. Errors or ambiguities in the natural language statements [15].

Examples from [15] clearly illustrate the significance of the validation test case scenario in detecting faulty statechart assertion diagram.

Consider the following requirement.

R5: An *event Q must occur within 30 seconds of every event P*. Figure 27 shows an invalid assertion diagram and Figure 28 shows a valid statechart assertion diagram respectively.

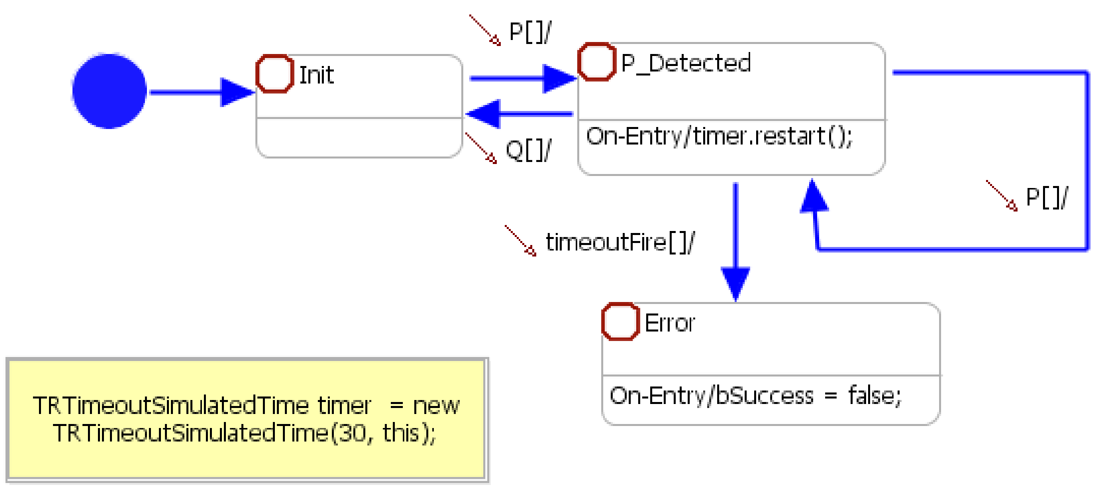

Figure 27: Invalid statechart assertion for R5 requirement. From [15].

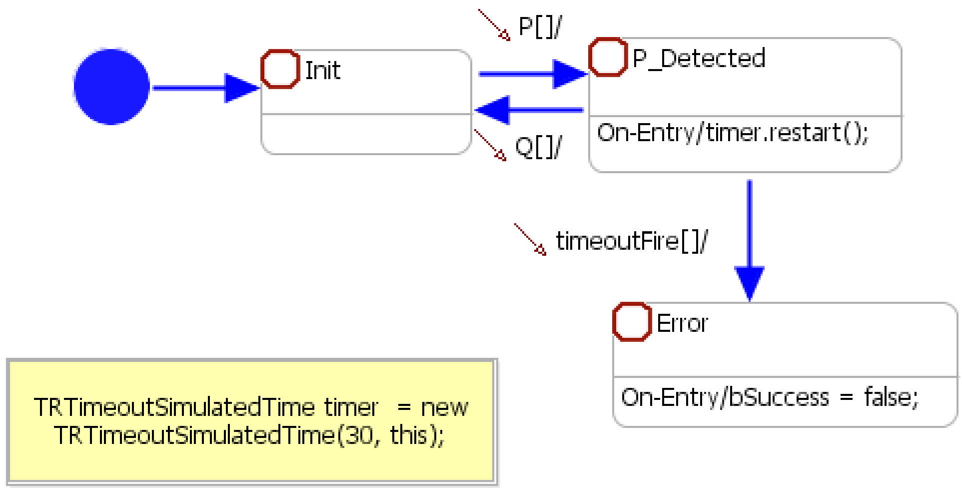

Figure 28: Valid statechart assertion for R5 requirement. From [15].

The validation test case presented in Listing 8 would provide the same results for both valid and invalid statechart assertion diagrams. However, another validation test case presented in Listing 9 would provide different results in these diagrams. It is important to notice that the test case in Listing 8 fails to detect the invalid statechart assertion in Figure 27. On the other hand, the validation test case in Listing 9 would detect the invalid statechart assertion in Figure 27. Specifically, the expectation for the validation of the test case in Listing 9 is to fail the assertion (i.e., bSuccess is false). However, the statechart assertion in Figure 27 would have bSuccess set to true, because loopback event P would reset the timer back to zero. Therefore, validation of the test case in Listing 9 would fail the test for the statechart assertion in Figure 27.

```
//test 1 - same result
sa.P();
sa.incrTime(20);
sa.Q()
assertTrue(sa.isSuccess());
```

Listing 8: Validation test case with the same result. From [15].

```
//test 2 - different result
sa.P();
sa.incrTime(20);
sa.P();
sa.incrTime(20);
sa.Q()  //Q occurs 40 seconds after first P
assertFalse(sa.isSuccess());
```

Listing 9: Validation test case with different results. From [15].

Statechart assertion validation serves two main purposes:

1. Resolve NL specification ambiguity and
2. Prune out errors in the assertion.

Furthermore, it is worthwhile to point out that, in any validation test case scenario, the tester always compares the test results with the behavior that is cognitively expected, based on the understanding of the requirement [9]. In case the behavior is not satisfied with the test results, then there are really three possibilities:

1. The assertion is an incorrect representation of the NL requirement [9].
2. The NL requirement is an incorrect or ambiguous representation of the cognitive requirement [9].
3. The cognitive requirement was not thought through well [9].

Validation for NL requirements in this thesis has concluded that both the MC Power Up Initialization Sequence and HHO were underspecified. However, since the MC system has already been developed and fielded, a few assumptions were made for the sake of progressing with this research.

V. VERIFICATION PHASE

The main purpose of the verification phase is to verify that the SUT's actual implementation conforms to the already-validated specifications. The approach taken in this thesis is to use log-file based verification, which is fully supported by StateRover. In contrast to most RV (Runtime Verification) monitor tools, log-file is a simpler and more scalable approach because of the following three reasons [4]:

1. It does not require closed loop architecture, whose use of inter-process communications mechanism such as TCP/IP would cause inevitable impact on the real-time characteristics of the real-time SUT. This impact would impede the corresponding tool's ability to truly verify the SUT's conformance to the time constraint requirement.

2. It works with almost any SUT that contains a file system. In fact, techniques being used in this thesis demonstrate that it could even work with an SUT without a file system.

3. It preserves the real-time characteristics of most SUTs [4].

Runtime verification was conducted in the EA-6B software development and testing lab environment. The lab setup is shown in the Figure 29. It closely resembles the actual EA-6B system architecture shown previously in Figure 8. All lab subsystems (i.e., TDS, IM, MM, MC, DSMU and MIDS) are actual avionics components suited for the flight on the airplane. This lab setup attempts to create a testing environment as close as possible to the actual EA-6B aircraft. The Host Workstation is used to communicate with MC SUT. It provides telnet and FTP (File Transfer Protocol) services, allowing the tester to communicate directly with the MC and also provides the ability to download log-file data from the MC. A Link-16 network requires at least two MIDS participants (i.e., MIDS terminals). One MIDS terminal is controlled by MC, which is the SUT for this thesis. The other MIDS terminal is controlled by a TJHS (TADIL (Tactical Digital Information Link) J Host Simulator), which simply acts as another MC with its own

MIDS terminal. Two MIDS terminals are connected via an RF (Radio Frequency) cable to prevent high intensity radiation in the lab environment.

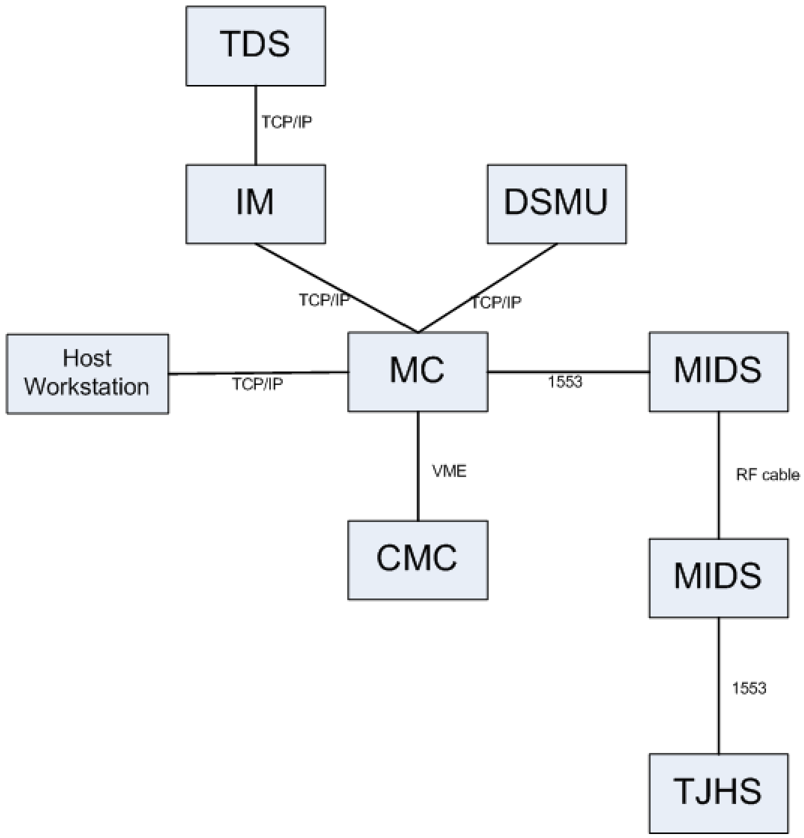

Figure 29: Lab setup for SUT verification

SUT verification consists of the following steps:

- Source code instrumentation using the StateRover's instrumentation plug-in.
- MC application re-build (using the instrumented source code files) with native MC software development environment.
- Executing the MC application on the target PowerPC system; log files are generated and uploaded to the Host Workstation via FTP server.
- Log file import into the Eclipse workspace using the StateRover's log file import tool. This step includes an automatic translation of the log file into a

corresponding JUnit file. This thesis denotes this JUnit test case as a verification JUnit test. This is to distinguish it from the validation JUnit tests discussed in Specification and Validation section.

● Namespace mapping, where the namespace used in statechart and propositional logic assertions of the assertion repository is mapped to the MC namespace (i.e., the namespace of methods and variables captured in the log file).

● Execution of the verification JUnit test within the assertion repository workspace [16].

A StateRover source code instrumentation plug-in is used to instrument SUT source code with snippets that enable the writing of data to a log file. The primary artifact being logged by the instrumentation tool is method calls (including arguments and time of execution), with the additional option of logging of variable assignments and source code branching decisions [4]. Figure 30 shows instrumentation plug-in with instrumented MC code. The Eclipse-based instrumentation plug-in tool does not require that the SUT be built or executed within the Eclipse IDE. The source code files were imported into Eclipse solely for the purpose of instrumentation. Once instrumented, the source code files were subsequently put into MC's native Tornado/VxWorks development environment and built in that environment.

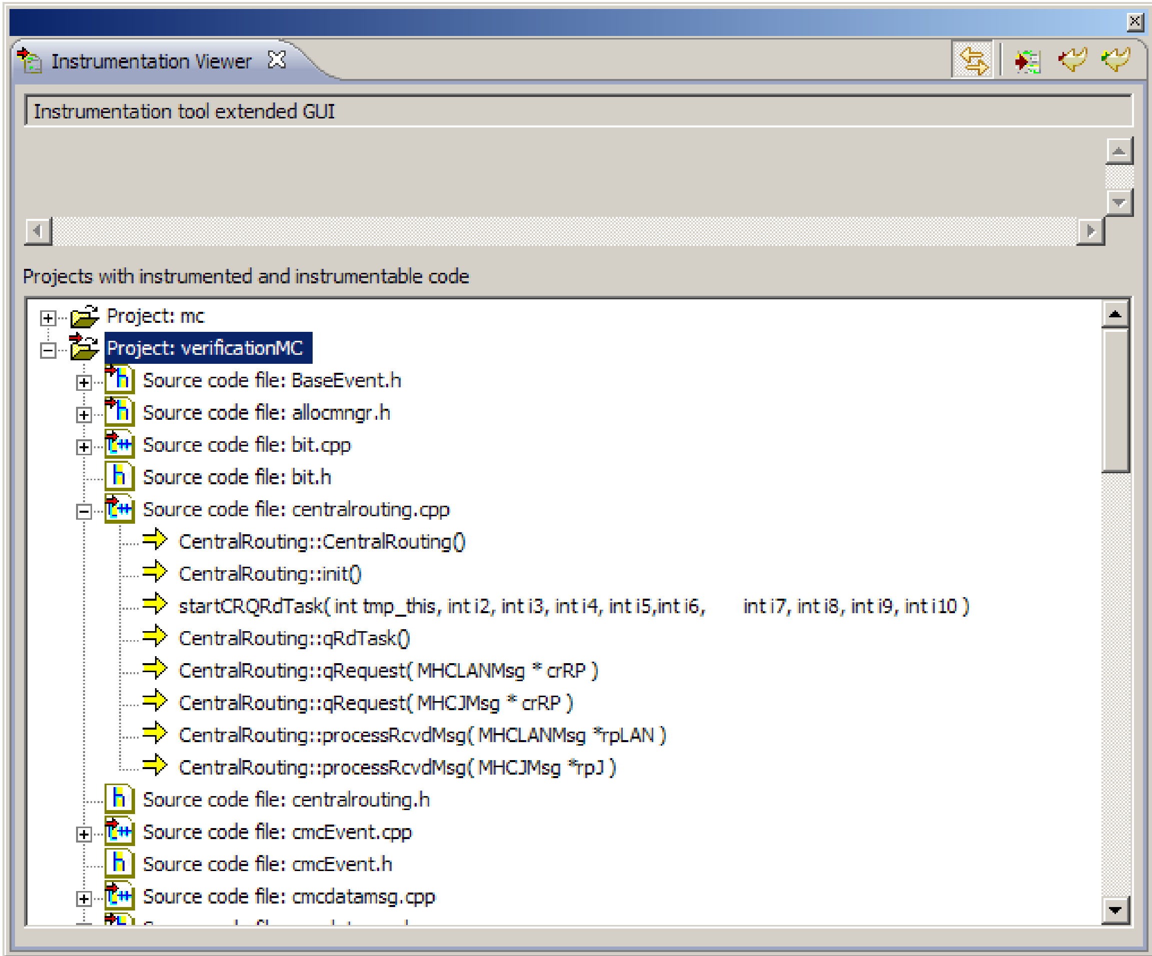

Figure 30: Instrumented MC code

Listing 10 shows a code snippet example for the instrumented periodicTask() functions.

Instrumentation has inserted MAYA_INSTRUMENT function capturing function signature along with location in the file. During run time when this function is executed, the MAYA_INSTRUMENT function would capture and timestamp this event.

```
void TerminalCtrl::periodicTask()
{
   int                  iCycleCount   = 0;
   int                  cmcCycleCount = 0;
   unsigned int         mstrRadValidFlags01, mstrRadValidFlags02;
   MHCLANMsg            *mhcLANMsg, *mstrRadLANMsg;
   MHCLANMsg            *rpLAN;
   STATUS               result = OK;

   /*  @instrumented Maya-Software. */
MAYA_INSTRUMENT((char
*)"<sig><![CDATA[TerminalCtrl::periodicTask()]]></sig>,"(char
*),""(char
*)"<sourcefile><![CDATA[C:/develop/MSSEwork/Thesis/StateChartWorkspace/
mcWorkSpace/verificationMC/terminalctrl.cpp]]></sourcefile>,"(char
*)"<pos val=\"47965\" instrval=\"54507\" />");
pCMC = new  CMCDATAMsg;
   CMCDATAMsg *      airStatus_CMC = new CMCDATAMsg;
   airStatus_CMC->id  = CMCIN_AIRCRAFT_STATUS;
   CMCin_AircraftStatus airStatus;

   d1553StatusInfo_t statusInfo;

   int tickDelayBeforeStart =  sysClkRateGet() / 20 - TC_DELAY_TICK;
   periodicTaskStarted = true;
```

Listing 10: Code snippet for instrumented PeriodicTask function

Instrumentation always places the MAYA_INSTRUMENT in the beginning of the function. Thus, the act of invoking and executing a function signifies a certain event.

However, one has to be aware of some instances when simply placing an instrumentation snippet in the beginning of the function will not produce the desired effect. For example, some function calls could be an entry point into a repeated task (or thread). The following example uses binary semaphores. The VxWorks operating system utilizes binary semaphores via system calls: SemTake and SemGive. These semaphores are used primarily for mutual exclusion to protect resources from being accessed from multiple tasks, and it is also used for task synchronization. The application function could call SemTake and get immediately blocked on this system call. In this case, the act of calling a function is not really a valid event, because the function had been immediately blocked. The valid event would be considered when some other function calls SemGive, thus unblocking the function from the SemTake call, and allowing it to proceed with its execution. The example is illustrated with a code snippet in Listing 11. In this example in

the getNavigationData function, the instrumentation snippet is placed right after the function definition. Thus, calling the function is not really an event, but rather simply an initialization call into a continuous task. However, in this case, it would be captured as an event. Listing 12 corrects the problem by placing the instrumentation snippet right after the SemTake function. The big challenge of course is that the correct placement of the instrumentation snippet now becomes dependent not on the syntax of the code, but rather on the semantic of the problem that the code is trying to implement. For the most part, this is not really an obstacle, but something to be aware of, in case the verification test produces unexpected results.

```
void NavController::getNavigationData()
{
  MAYA_INSTRUMENT((char
*)"<sig><![CDATA[NavController::getNavigationData()]]></sig>,"(char
*),""(char
*)"<sourcefile><![CDATA[C:/develop/example.cpp]]></sourcefile>,"(char
*)"<pos val=\"47965\" instrval=\"54507\" />");

  while(true) //execute forever as a task
  {
      SemTake(); //block here and wait for other task to call SemGive
      for (int i=0; i < maxValu; i++)
      {
        calculateNavSolution();
      }
  }
}
```

Listing 11: Instrumentation is placed prior to SemTake system call (incorrect)

```
void NavController::getNavigationData()
{

  while (true) //execute forever as a task
  {
    SemTake(); //block here and wait for other task to call SemGive

    MAYA_INSTRUMENT((char
*)"<sig><![CDATA[NavController::getNavigationData()]]></sig>,”(char
*),””(char
*)"<sourcefile><![CDATA[C:/develop/example.cpp]]></sourcefile>,”(char
*)"<pos val=\"47965\" instrval=\"54507\" />");
    for (int i=0; i < maxValu; i++)
    {
      calculateNavSolution();
    }
  }
}
```

Listing 12: Instrumentation is placed after SemTake system call (correct)

After instrumentation of MC source code files, the Tornado IDE is used to build (compile and link) the MC executable file. StateRover provides three stock files that have to be compiled and linked along with the SUT files. These three files are used for capturing and uploading all encountered events while SUT is running on the target. In order to minimize any intrusion on the SUT and to provide a higher degree of control and flexibility, the stock files have been modified to accommodate the SUT runtime environment. The approach for capturing a verification log-file is to buffer/store all events into memory as soon as events are generated while the SUT is executing on the target PowerPC computer. Then, upon execution of a specific upload command from the tester, the entire buffer is uploaded into the Host Workstation FTP server. Writing events into memory is insignificant, taking only a few microseconds with no impact on real-time performance. Also, the tester has finer control on when to start or stop capturing events, and when to actually upload events into the FTP server. It is important to note the flexibility and non-intruding aspect of a log-file based approach. In fact, the log-file could even be captured not only in the lab but on the actual EA-6B airplane in flight. In that case, the stock file could be modified to start capturing events at certain time or by keying off of some start message. Then the event data could be transferred into the PowerPC file based flash non-volatile memory or uploaded to the DSMU, which acts as an NFS (Network File System) server.

After the system is loaded and ready for the verification log, the tester has direct access to the SUT via a telnet interface. The log-file for the MC Power Up Initialization Sequence was captured right after system power up. The log-file is an XML based file, which is ideal for readability, transport between computers, and text string manipulation and processing. Listing 13 depicts a snippet of the actual captured XML file for the MC Power Up Initialization Sequence. Listing 14 shows a snippet of the actual XML file for HHO. It is interesting to note that the MC Power Up Initialization Sequence log-file starts with an event having a 0 seconds time stamp. On the other hand, the HHO log-file starts with an event having a 430 seconds time stamp. This indicates that event capturing had started at 430 seconds since MC power up. The log-file is a cumulative collection of test runs. This means it could contain a number of different test runs, where each test run is simply appended to the log-file.

```
"<newtest>"
<event>
<sig><![CDATA[IMInterface::init()]]></sig>
<time lang="c" unit="sec" val="0" />
</event>
<event>
<sig><![CDATA[StatusRpt::init()]]></sig>
<time lang="c" unit="sec" val="0" />
</event>
<event>
<sig><![CDATA[TerminalCtrl::init()]]></sig>
<time lang="c" unit="sec" val="0" />
</event>
<event>
<sig><![CDATA[BuiltInTest::init()]]></sig>
<time lang="c" unit="sec" val="0" />
</event>
```

Listing 13: Log file snippet for MC Power Up Initialization Sequence

```
<newtest>
<event>
<sig><![CDATA[TerminalCtrl::snd1553_BIMA01()]]></sig>
<time lang="c" unit="sec" val="430" />
</event>
<event>
<sig><![CDATA[TerminalCtrl::snd1553_BIMA04()]]></sig>
<time lang="c" unit="sec" val="430" />
</event>
<event>
<sig><![CDATA[CMCInterface::getReadDataReadyFlag( map<unsigned short,
cmc_data_info>::iterator& pos )]]></sig>
<time lang="c" unit="sec" val="430" />
</event>
<event>
<sig><![CDATA[TerminalCtrl::get1553_BOMA01()]]></sig>
<time lang="c" unit="sec" val="430" />
</event>
```

Listing 14: Log file snippet for HHO

The MC Power Up Initialization Sequence log-file contains about 4350 events captured in the span of 54 seconds. HHO log-file contains about 11825 events captured in the span of 55 seconds.

The XML based log-files were imported into the assertion repository by using the StateRover input plug-in. StateRover converts a log-file from XML format into an equivalent verification JUnit test format. Listing 15 shows a snippet of the converted verification JUnit test corresponding to the MC Power Up Initialization Sequence XML log-file shown in Listing 13. Similarly, Listing 16 shows a snippet of the converted verification JUnit test corresponding to HHO XML log-file shown in Listing 14.

```
public void testMe1() throws Exception {
assertions.reset("assertionrepository.PowerUpSequence_namespace_map");
assertions.setAnimationMode(IAnimationStatusConstants.ANIMATION_MODE_CO
VERAGE_SINCE_CONSTRUCTION);

       nBaseTime = (int)(0L  * nFactor_LogUnitsAreSec);
       assertions.fire("IMInterface::init()");

       assertions.incrTime((int)(0L  * nFactor_LogUnitsAreSec -
nBaseTime));
       nBaseTime = (int)(0L  * nFactor_LogUnitsAreSec);
       assertions.fire("StatusRpt::init()");

       assertions.incrTime((int)(0L  * nFactor_LogUnitsAreSec -
nBaseTime));
       nBaseTime = (int)(0L  * nFactor_LogUnitsAreSec);
       assertions.fire("TerminalCtrl::init()");

       assertions.incrTime((int)(0L  * nFactor_LogUnitsAreSec -
nBaseTime));
       nBaseTime = (int)(0L  * nFactor_LogUnitsAreSec);
       assertions.fire("BuiltInTest::init()");
```

Listing 15: Verification JUnit test for MC Power Up Initialization Sequence

```
public void testMe1() throws Exception {
assertions.reset("assertionrepository.HHO_namespace_map");
assertions.setAnimationMode(IAnimationStatusConstants.ANIMATION_MODE_CO
VERAGE_SINCE_CONSTRUCTION);

    nBaseTime = (int)(430L  * nFactor_LogUnitsAreSec);
    assertions.fire("TerminalCtrl::snd1553_BIMA01()");

    assertions.incrTime((int)(430L  * nFactor_LogUnitsAreSec -
nBaseTime));
    nBaseTime = (int)(430L  * nFactor_LogUnitsAreSec);
    assertions.fire("TerminalCtrl::snd1553_BIMA04()");

    assertions.incrTime((int)(430L  * nFactor_LogUnitsAreSec -
nBaseTime));
    nBaseTime = (int)(430L  * nFactor_LogUnitsAreSec);
 assertions.fire("CMCInterface::getReadDataReadyFlag(map<unsigned
short,cmc_data_info>::iterator& pos)");

    assertions.incrTime((int)(430L  * nFactor_LogUnitsAreSec -
nBaseTime));
    nBaseTime = (int)(430L  * nFactor_LogUnitsAreSec);
    assertions.fire("TerminalCtrl::get1553_BOMA01()");
```

Listing 16: Verification JUnit test for HHO

The next step in the MC verification process was to match the namespace used by the assertions to the C++ namespace of the MC code base, namely, to the namespace of the method calls logged in the log-file. This mapping is done using a namespace mapping GUI, part of the StateRover's namespace plug-in. This GUI allows manual and algorithmic mapping (using built-in as well as customizable mapping algorithms) of the two namespaces. For instance, matching algorithms such as "90% LCS" declares a match between string *r* (the log-file string) and *s* (an assertion event name string) if the length of the longest common subsequence (LCS) of *r* and *s* is greater than 0.9 times the length of *s* and is also greater than 0.5 times the length of *r* [4]. An instance namespace map used for MC verification is depicted in Figure 31.

For example, it maps ProcessLoopbackResponse to Loopback_J12_0. The namespace mapping plug-in also generates Java code (denoted executable namespace translation) that implements this mapping [16].

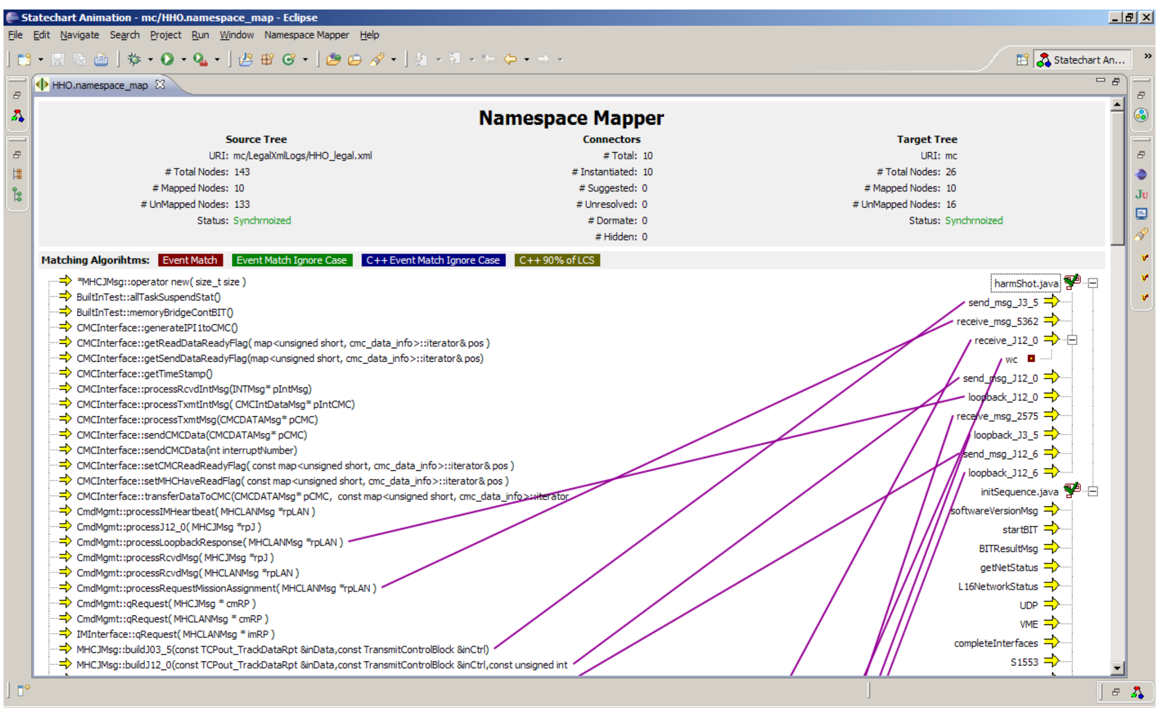

Figure 31: Namespace mapping

The final step in verification is to run JUnit test in the assertion repository. Verification operates as follows:

Every event in the verification JUnit test (e.g., assertions.fire("TerminalCtrl::init()"); in Listing 15) corresponds to a method that fired and subsequently logged. This event is translated, using the executable namespace translation code, into the namespace of the assertion repository, and then dispatched to all assertions in the repository. Every assertion that contains that event responds by possibly changing states, while all others simply ignore it [16].

Verification test for the MC Power Up Initialization Sequence had failed time constraint of 10 seconds. Figure 32 depicts these verification test results.

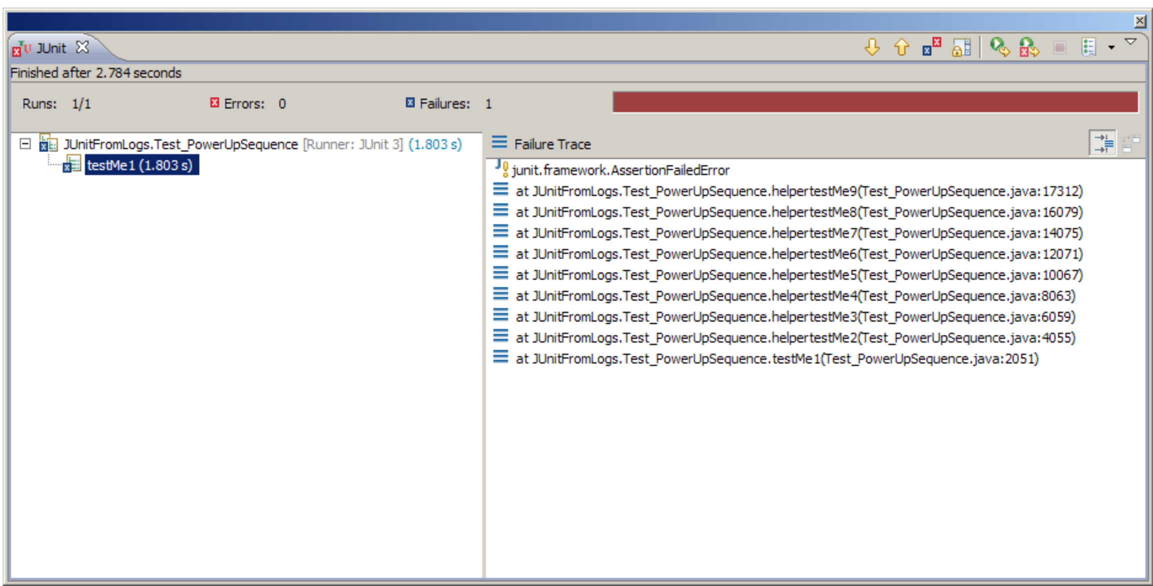

Figure 32: Verification test results for MC Power Up Initialization Sequence

The assertion repository also keeps track of and reports the name of all assertions that failed (bSuccees = false) during verification JUnit test and all SUT fired events with no mapping to an assertion event. No mapping between SUT fired events and assertion events does not necessarily constitutes an error. It simply means that SUT fired events were not part of the statechart assertion repository and thus outside the scope of the verification test. However, these unmatched events should still be analyzed, since this

could indicate an error in the namespace mapping. Figure 33 shows the assertion repository viewer for the MC Power Up Initialization Sequence.

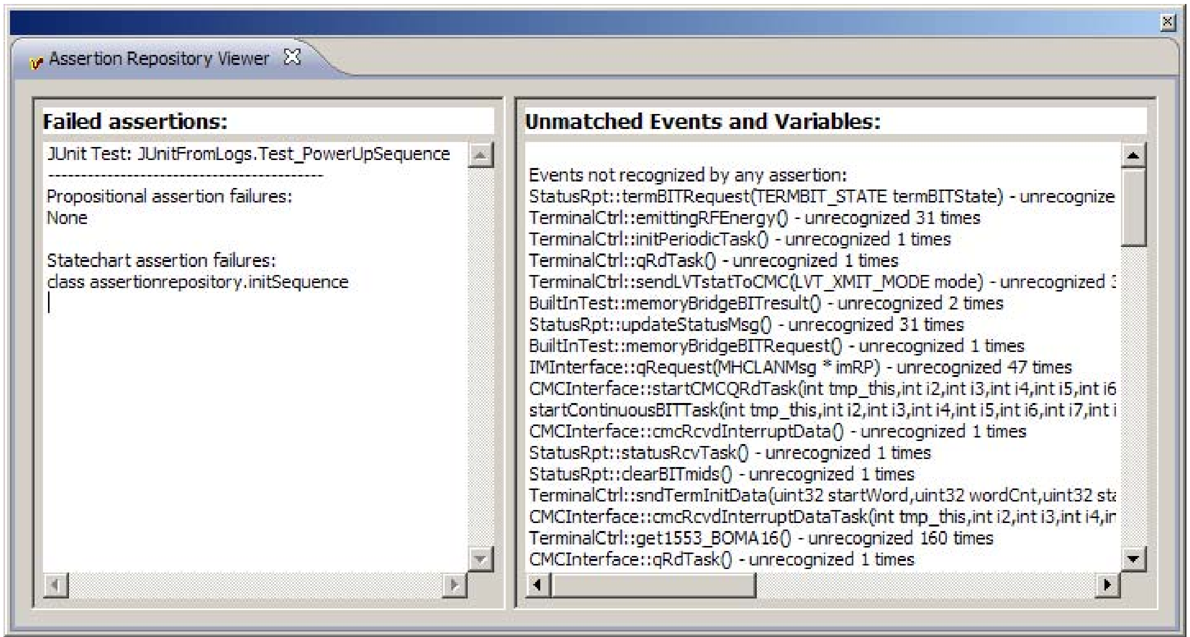

Figure 33: MC Power Up Initialization Sequence with failed assertion and unmatched events

The time constraint failure is evident from coverage animation depicted in Figure 34.

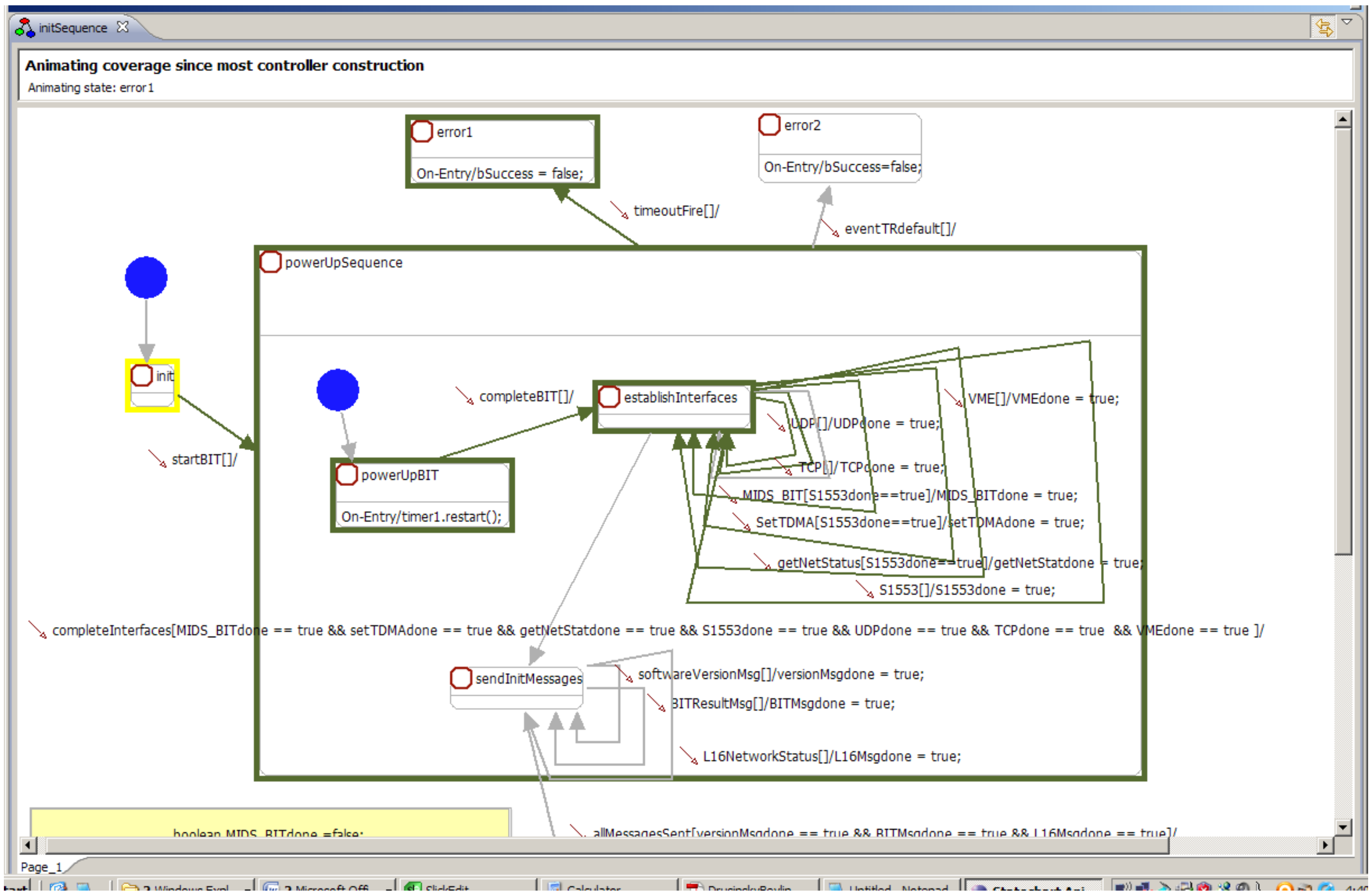

Figure 34: Coverage animation from verification test for failed MC Power Up Initialization Sequence

From Figure 34 it is evident that timeoutFire caused a transition to error1 state. Due to a timeout transition, a few events were not traversed, as shown in Figure 34. In particular, MIDS_BIT event was not traversed, which caused a failed transition to the sendMessages state. The failed transition to the sendMessage state is attributed to the conditional completeInterfaces event, which is dependent on the prior occurrence of the MIDS_BIT event. From Figure 34, it is clear that MIDS_BIT event is the root cause of verification failure. The timeline diagram depicted in Figure 35 provides evidence and data confirmation indicating MIDS_BIT occurring beyond the 10-second time constraint. According to Figure 35, MIDS_BIT occurred at 13 seconds, which is beyond the 10-second specification.

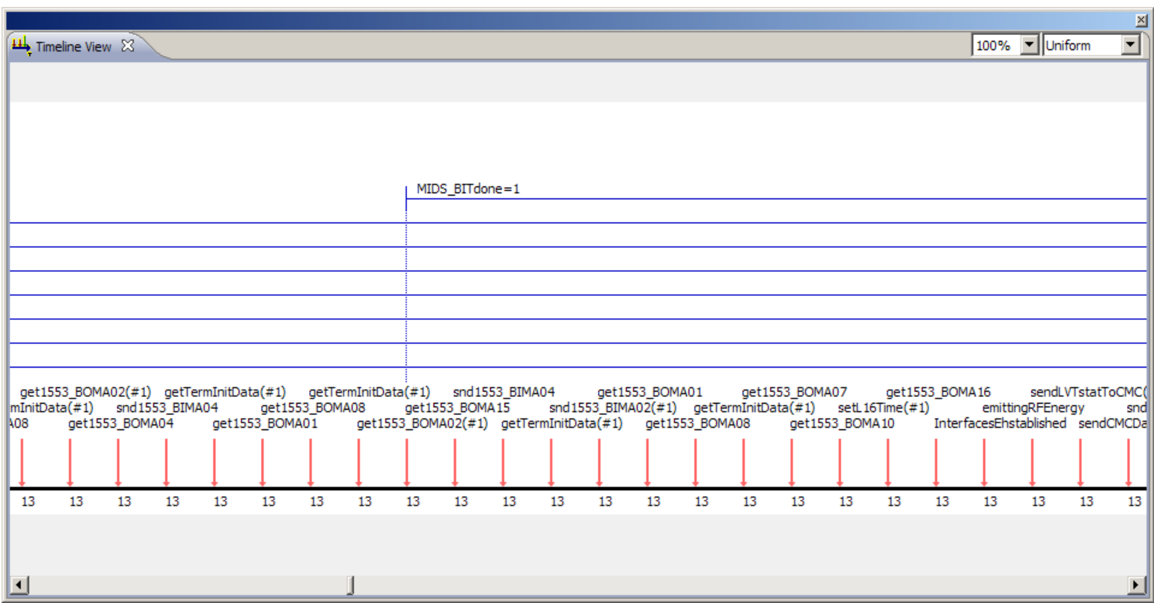

Figure 35: Timeline diagram for verification test (Power Up Initialization Sequence)

HHO had passed verification test. Figure 36, Figure 37, and Figure 38 show JUnit test results, assertion repository, and coverage animation, respectively.

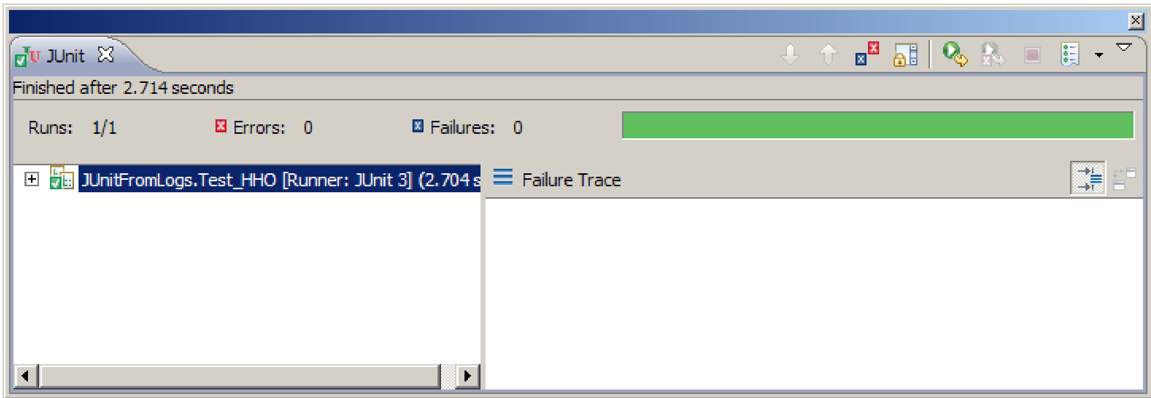

Figure 36: Verification test results for HHO

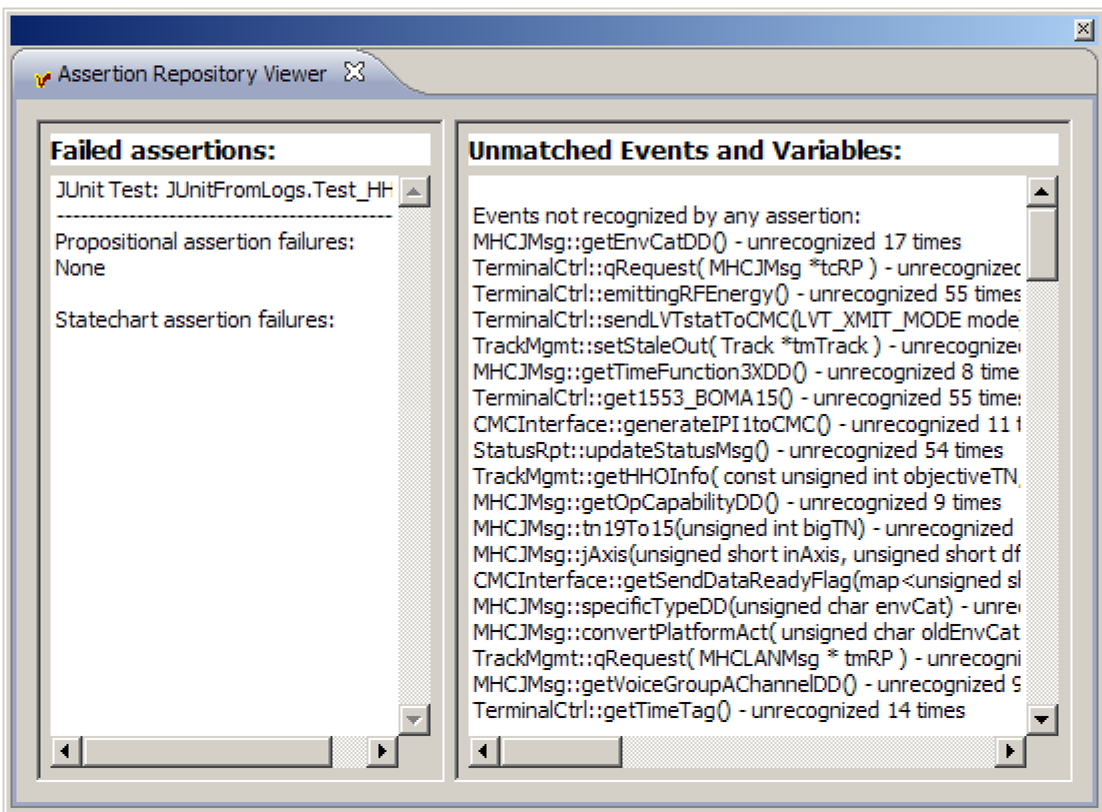

Figure 37: Assertion repository viewer for HHO

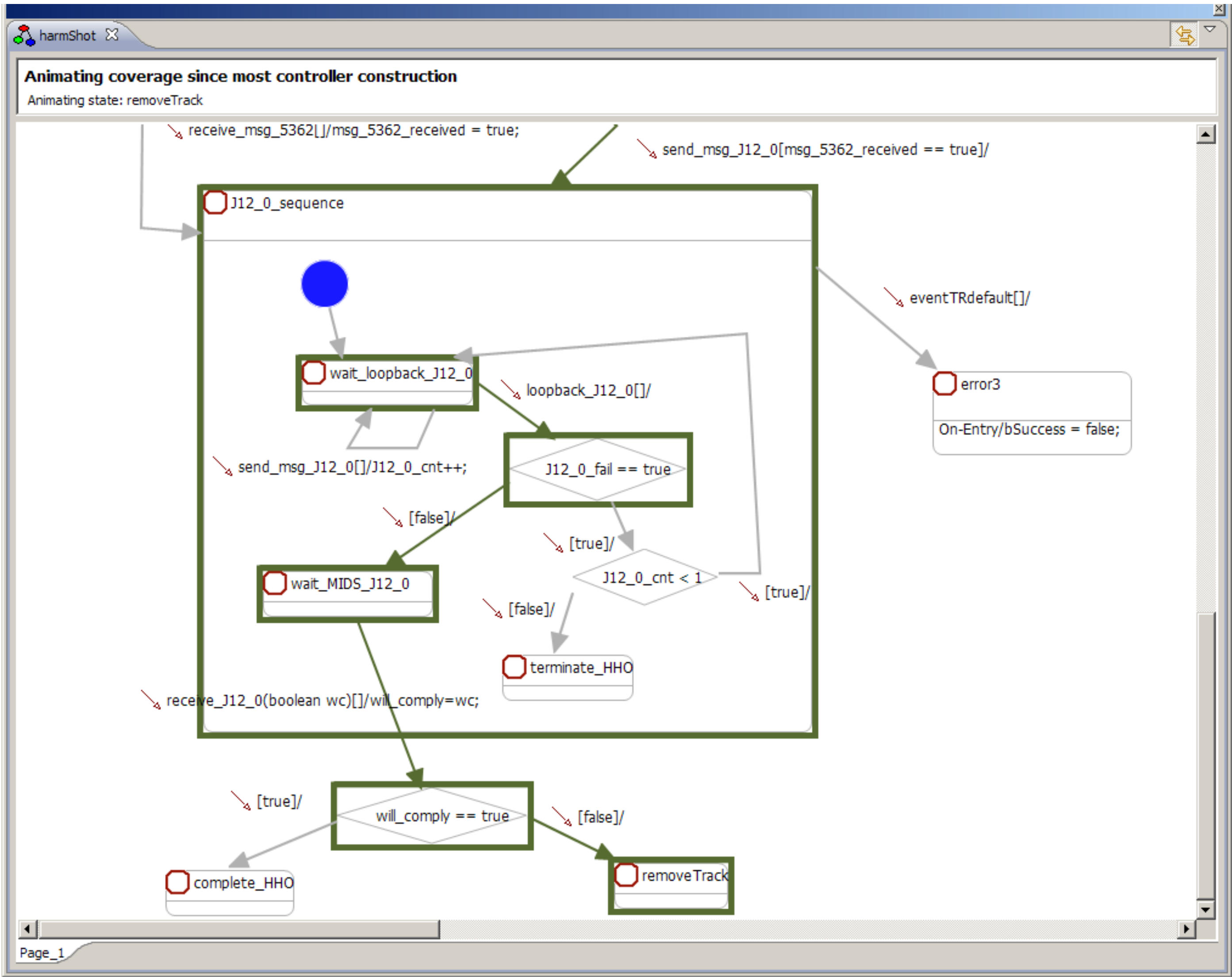

Figure 38: Animation coverage from verification test for HHO

Statechart assertion diagrams could be effectively used as a test artifact for coverage criterion. Figure 38, animation coverage, indicates that not all events and paths were covered (e.g., event send_msg_J12_0). For example, if coverage criterion for the verification test was to cover all events and logical paths, then more verification tests would be necessary to fully satisfy coverage criterion. Again, this emphasizes the value of coverage criterion as a practical guideline on when to stop testing.

VI. CONCLUSION

This thesis has demonstrated in practice the entire cycle of validation and verification applied to a real-time embedded system. It fully addressed three main thesis questions with a practical application of the specification, validation and verification techniques on the MC system using the StateRover tool.

1. Is it feasible to capture Natural Language (NL) requirements into formalized specification and subsequently validate these requirements?

It has been demonstrated that any verifiable NL requirements, regardless of their complexity, can be formalized into intuitive UML-based statechart assertion diagrams. Validation of specifications is accomplished by JUnit validation test cases run against the auto generated assertion repository. The process in itself of developing statechart assertion diagrams and subsequent validation has revealed various specification omissions and ambiguities.

2. Is it possible for the V&V team to really have an independent, fresh view of the system that matches the developer's cognitive intent of the requirement?

V&V team independence comes from the fact that the test team is not involved in the system design and implementation. In this way, the V&V team is not biased by any design or implementation (i.e., program code) artifacts. It is important that the V&V team should be involved right from the start of the project, validating specifications and making sure specifications match their cognitive intent as well as the developers' cognitive intent. In this way, the chances of specification omission or ambiguity are drastically reduced. For verification testing, actual code has to be instrumented. However, the V&V team does not need to have any insight into the implementation details. Additionally, instrumented code is actually built within the native development IDE. The V&V team might need some assistance with namespace mapping, to match log-file events and assertion repository events. However, there exist powerful matching algorithms to significantly reduce events matching task.

3. Is it possible to verify the system's sequential, temporal, and propositional behavior while running on an actual target system?

The thesis has demonstrated the entire log-file based verification process on the MC system. More importantly, it demonstrated that the log-file technique is non-intrusive and does not affect the system's temporal behavior. Also, the thesis has demonstrated the techniques' flexibility to adapt to any runtime environment, even if the SUT does not have a file system.

The V&V techniques presented in this thesis directly addressed and significantly improved three common problems with software testing: observability, natural language specification, and coverage criterion. Observability is achieved by capturing log-files at runtime and running JUnit verification tests. This provides direct insight into the SUT temporal behavior with detailed test results and animation coverage. The whole premise of the V&V technique presented in this thesis is based on capturing Natural Language Specification using the statechart assertion formalism. It is quite intuitive, flexible, and scalable. Given the fact that requirements could be volatile, the V&V techniques presented in this thesis and the StateRover tool allow for effective and rapid repeated validation and verification cycles, for example making any necessary modifications to a statechart assertion diagram due to specification changes, and then re-running validation testing.

A statechart assertion diagram could be used in selecting coverage criterion. After all, testing cannot be considered fully complete until all events and logic paths have been traversed.

The usefulness of the V&V techniques presented in this thesis has been readily demonstrated by detecting a failure of the MC system. MC is a fully operational and fielded system, therefore it is safe to assume that it has passed all the rigorous quality tests, and yet verification tests have found a time constraint failure.

Future research might focus on statistical assertions and automatic test generations, where assertions are created automatically using training data.

LIST OF REFERENCES

[1] National Institute of Standard and Technology, Gaithersburg, MD, “The economic impacts of inadequate infrastructure for software testing,” Planning Report 02-3, May 2002.

[2] IEEE, IEEE Std. 1012-2004, IEEE standard for software verification and validation, 2004.

[3] D. Drusinsky, J. B. Michael, and M. Shing, “The three dimensions of formal validation and verification of reactive system behaviors,” Naval Postgraduate School, August 2007.

[4] D. Drusinsky, “UML-based specification, validation, and log-file based verification of the Orion Pad Abort software,” Technical Report, Naval Postgraduate School, May 2010.

[5] D. R.Wallace and R. U. Fujii, “Software Verification and Validation an Overview,” IEEE, May 1989, pp. 10–17.

[6] P. Ammann and J. Offutt, Introduction to Software Testing, Cambridge University Press, 2008.

[7] Lecture Notes for Software Requirements Engineering, Naval Postgraduate School, July 2009.

[8] D. Drusinsky, J. B. Michael and M. Shing, “A framework for computer-aided validation,” Innovations in Systems and Software Engineering, vol.4, no. 2, June 2008, pp. 161–168.

[9] D. Drusinsky, Modeling and Verification Using UML Statecharts – A Working Guide to Reactive System Design, Runtime Monitoring and Execution-based Model Checking, Elsevier, 2006.

[10] H. Kopetz, R. Zainliger, G. Fohler, H. Hantz, P. Puschner, and W. Schutz, “The design of real-time systems: from specification to implementation and verification,” Software Engineering Journal, May 1991.

[11] D. Harel, “Statecharts: A visual formalism for complex systems,” *Science of Computer Programming*, vol.8, no. 3, 1987, pp. 231–274.

[12] M. Huth and M. Ryan, Logic in Computer Science, Modeling and Reasoning about Systems, Second Edition, Cambridge University Press 2004.

[13] D. Drusinsky, M. Shing, and K. Demir, "Creating and validating embedded assertion statecharts," *IEEE Distributed Systems Online*, vol. 8, no. 5, 2007, art. no. 0705-o5003.

[14] Space and Naval Warfare Systems Command (SPAWAR), "Concept of Operations, Integration Link-16 into the EA-6B aircraft," February, 2001.

[15] D. Drusinsky, B. Michael, T. Otani, and M. Shing, "Validating UML Statechart-Based Assertions Libraries for Improved Reliability and Assurance," *Proceedings of the Second International Conference on Secure System Integration and Reliability Improvement* (SSIRI 2008), Yokohama, Japan, 14–17, July 2008, pp. 47–51.

[16] D. Drusinsky and C. Beylin, "Assertion-based formal Specification, Validation and Verification of the MIDS Controller," *22nd IEEE International Symposium on Rapid System Prototyping*, Karlsruhe, Germany, May, 2011.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Balaji Iyer
   Naval Air Warfare Center Weapons Division, code 45P100E
   Point Mugu, California

4. Doron Drusinsky
   Naval Postgraduate School
   Monterey, California

5. Man-Tak Shing
   Naval Postgraduate School
   Monterey, California

6. Dr. Butch Caffall
   NASA Goddard Space Flight Center
   Greenbelt, Maryland

7. Mr. Steve Raque
   NASA IV&V Facility
   Fairmont, West Virginia